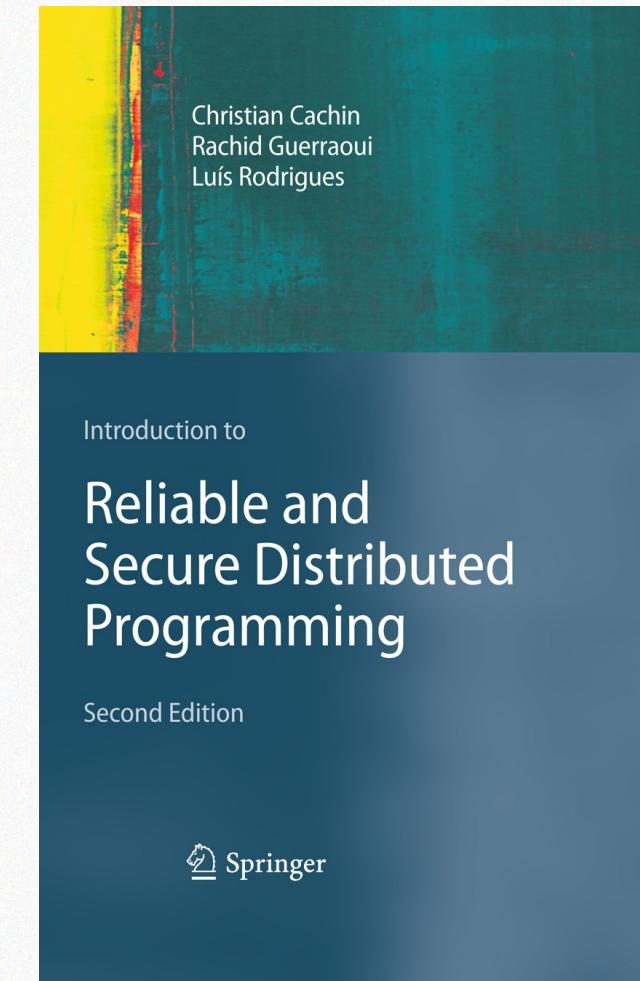
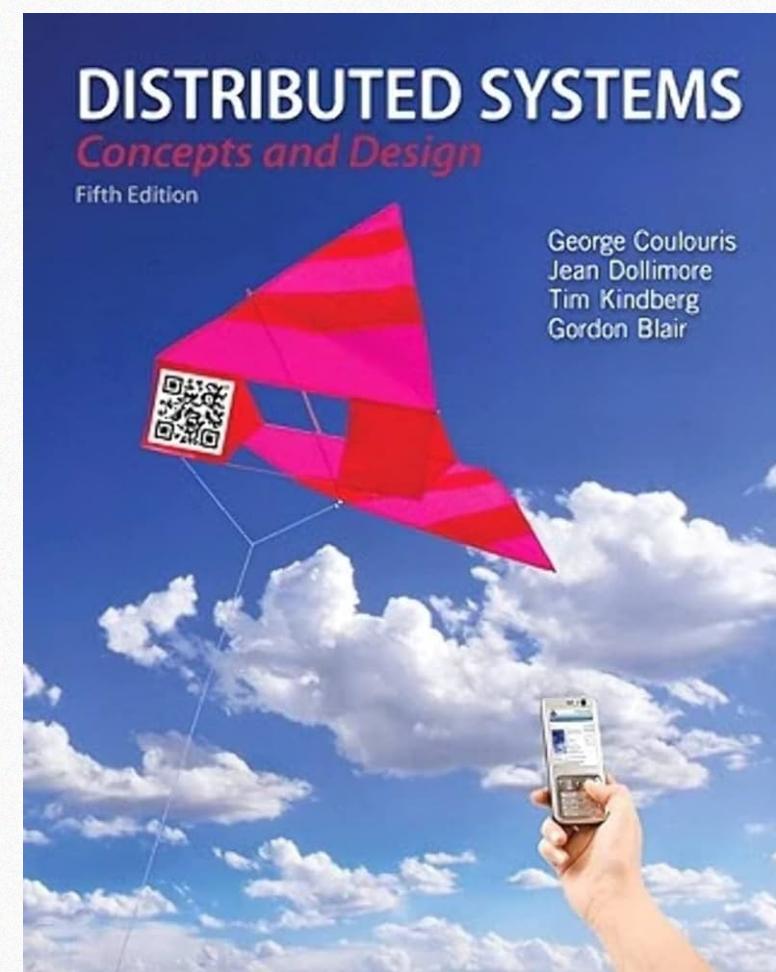
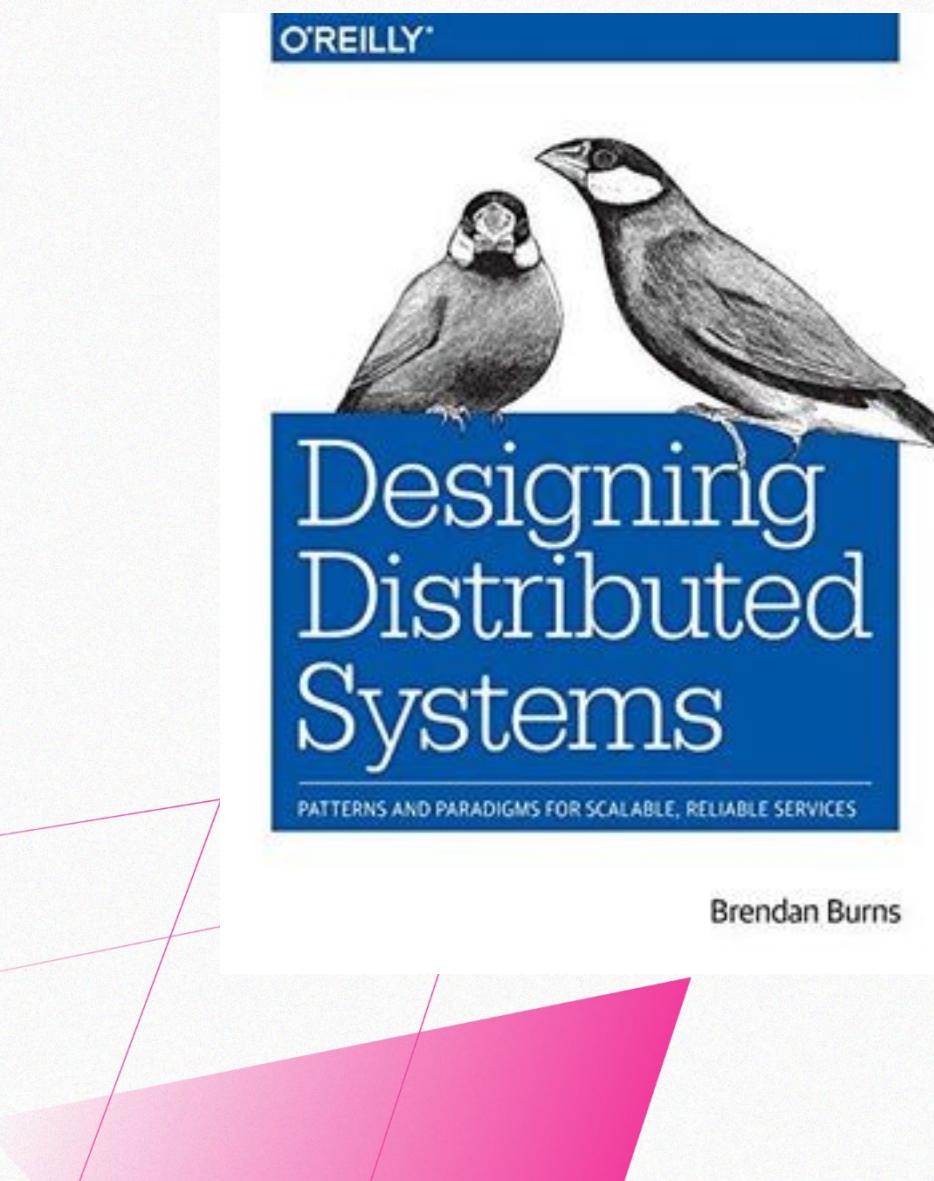


Arquitectura de Sistemas Distribuidos

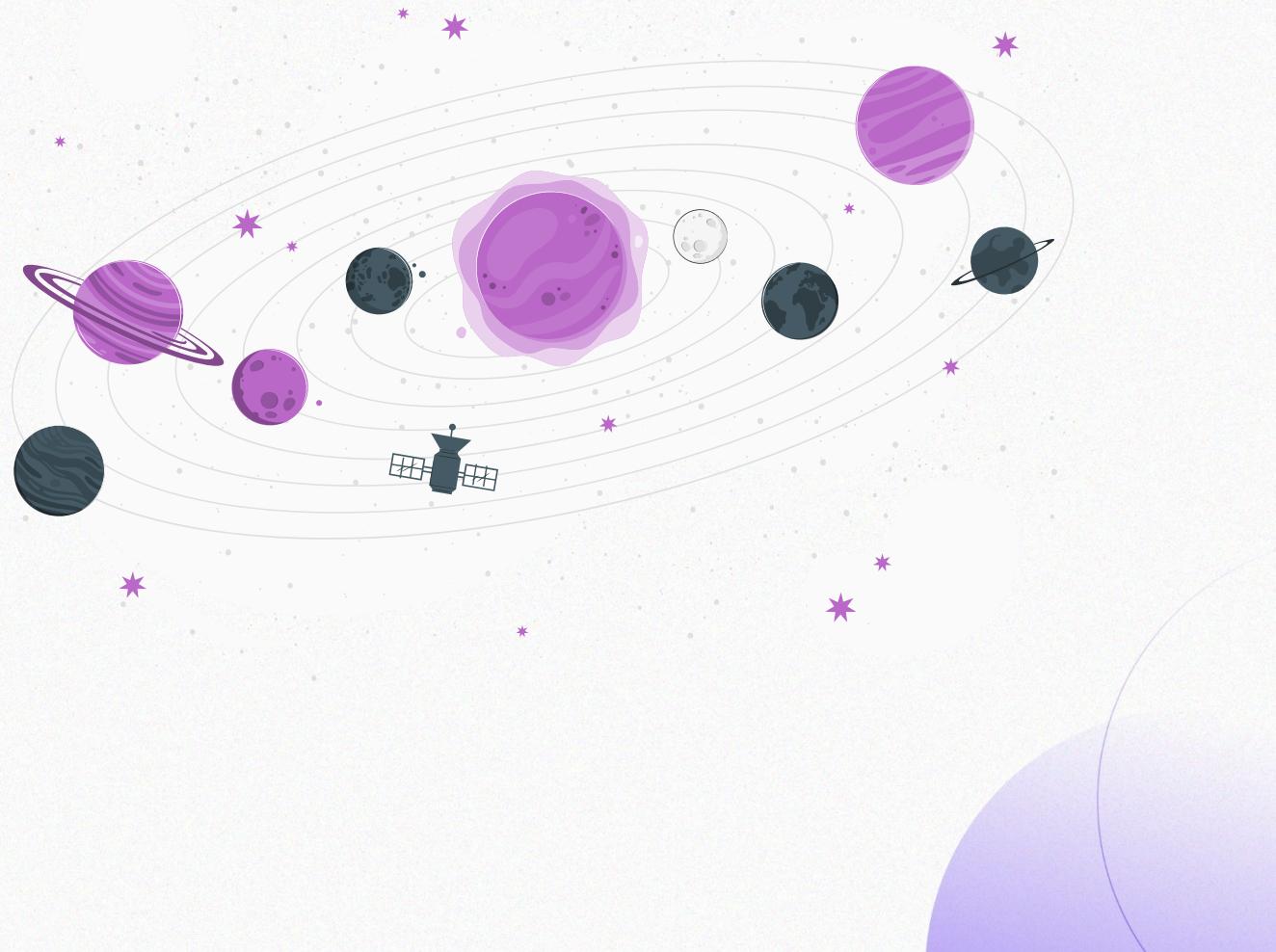
“El diseño arquitectónico de software es la clave para construir sistemas sostenibles.”

— Grady Booch



¿Qué es un sistema distribuido?

Un sistema distribuido es, en esencia, un conjunto de computadoras independientes que trabajan juntas para aparentar ser un único sistema cohesivo. Para lograrlo, estas máquinas comparten recursos, datos y tareas, a menudo a través de una red.



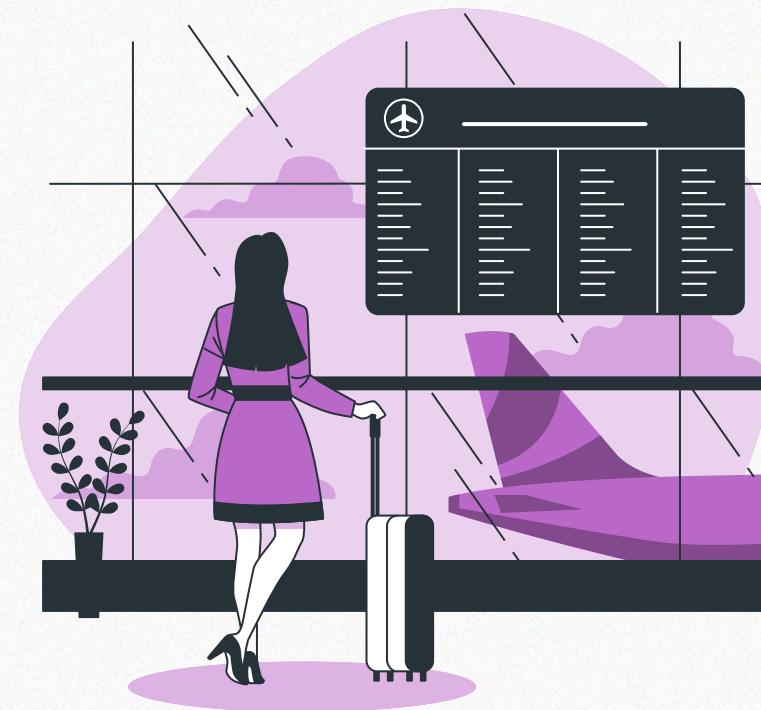
Ejemplo sencillo:

Pensemos en un sistema de reserva de vuelos.
¿Qué ocurre cuando haces una búsqueda?

Los datos que ves provienen de múltiples servidores:

- Uno que almacena información de vuelos.
- Otro que maneja las reservas.
- Uno adicional que verifica la disponibilidad de asientos en tiempo real.

Aunque todo esto parece sencillo desde el punto de vista del usuario, detrás hay una compleja interacción entre múltiples máquinas distribuidas geográficamente.



Problemas que los sistemas distribuidos buscan resolver

1. Escalabilidad:

A medida que aumenta la cantidad de usuarios, un sistema centralizado simplemente no puede manejar toda la carga. Los sistemas distribuidos permiten dividir el trabajo entre múltiples máquinas, haciendo posible que plataformas como **YouTube** manejen miles de millones de peticiones diarias.

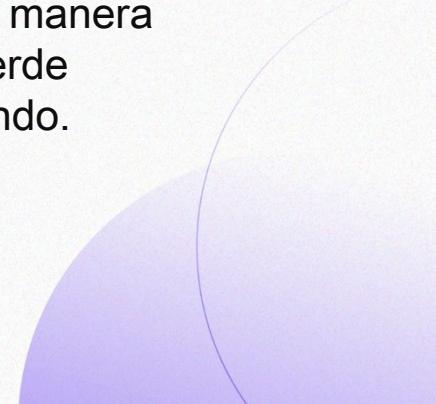


2. Confiabilidad:

¿Qué pasa si un servidor falla? En un sistema distribuido bien diseñado, otro servidor puede tomar su lugar, asegurando que el sistema siga funcionando. Esto es especialmente importante para servicios críticos, como plataformas bancarias.

3. Disponibilidad:

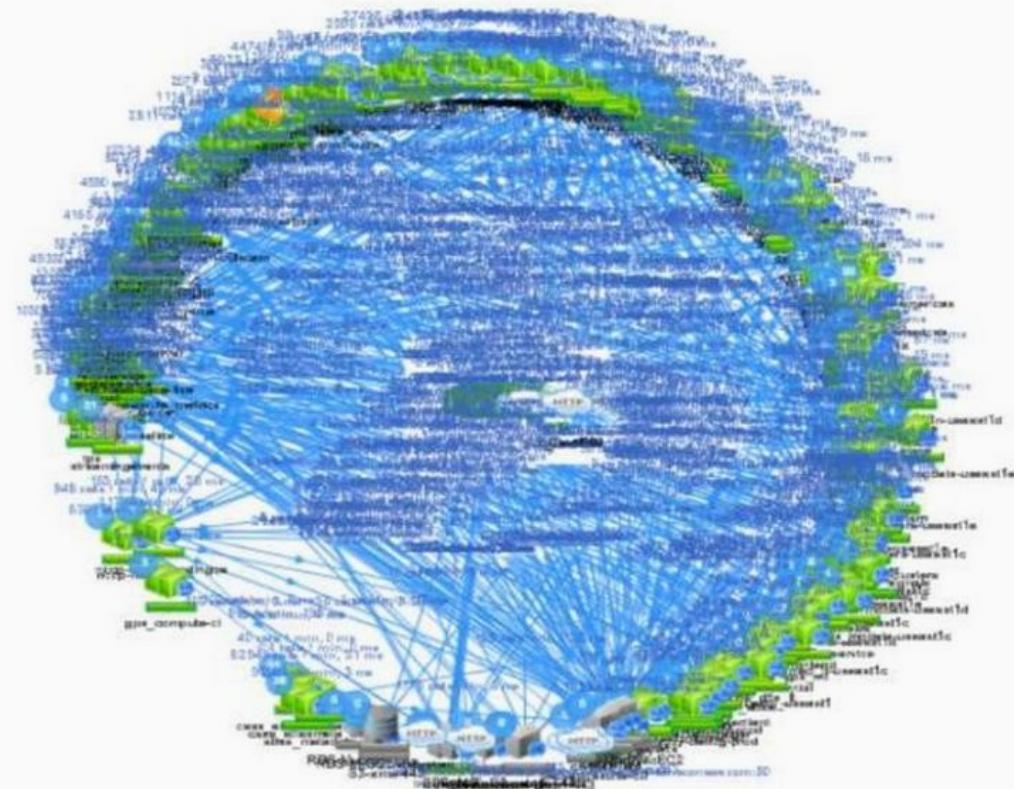
Los usuarios esperan que los servicios estén disponibles **24/7**. Los sistemas distribuidos permiten replicar datos y servicios, de manera que incluso si una región completa pierde conexión, otra pueda continuar operando.



Ejemplos de sistemas distribuidos en el mundo real

1. Netflix:

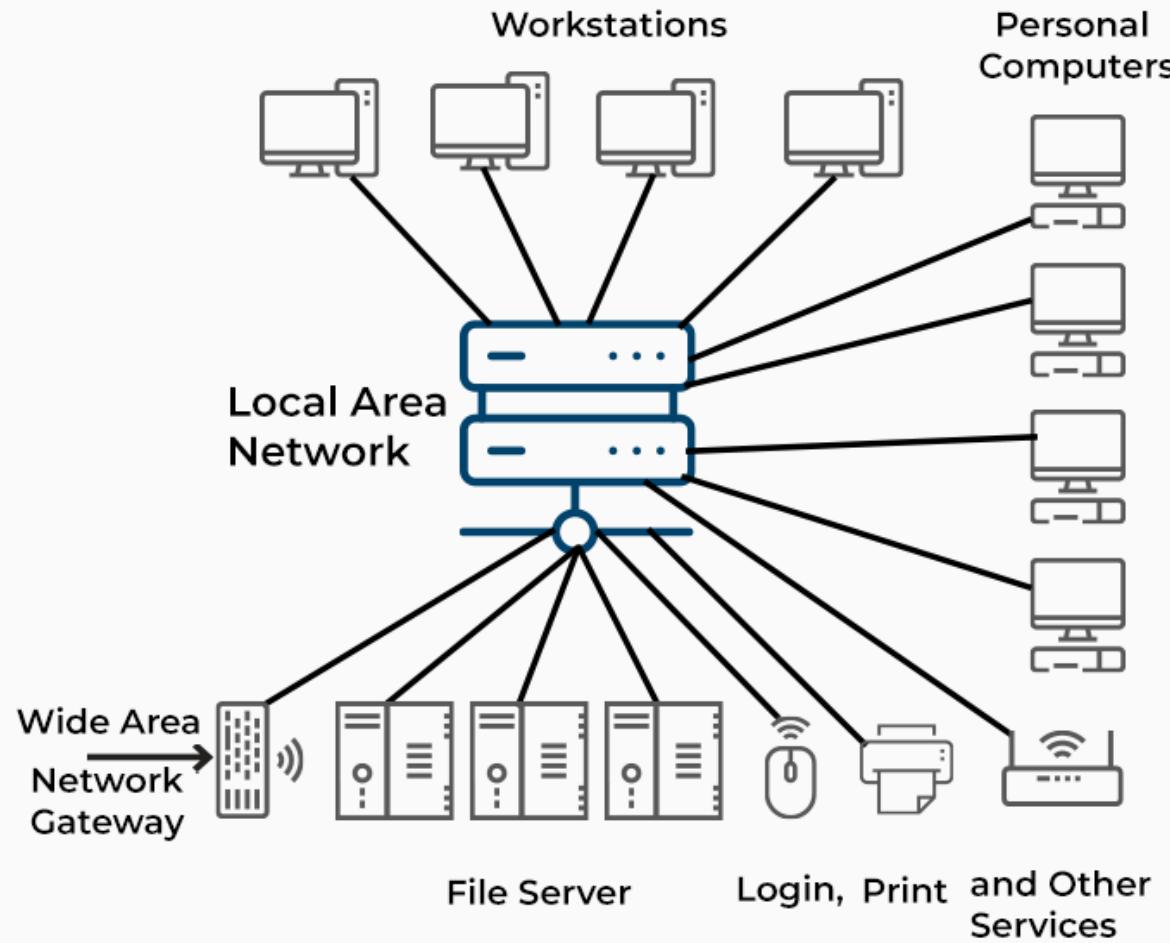
Netflix utiliza sistemas distribuidos para almacenar y entregar contenido en tiempo real a usuarios de todo el mundo. Utilizan redes de distribución de contenido (CDNs) y servidores ubicados estratégicamente para reducir la latencia.



Desafíos fundamentales en los sistemas distribuidos

A pesar de sus beneficios, diseñar un sistema distribuido es extremadamente complejo. Algunas de las preguntas clave que enfrentan los arquitectos son:

- **Sincronización:** ¿Cómo garantizar que las máquinas estén de acuerdo en el estado del sistema?
- **Fallas:** ¿Qué hacemos cuando un servidor no responde?
- **Consistencia:** ¿Cómo aseguramos que los datos sean los mismos en todas partes?

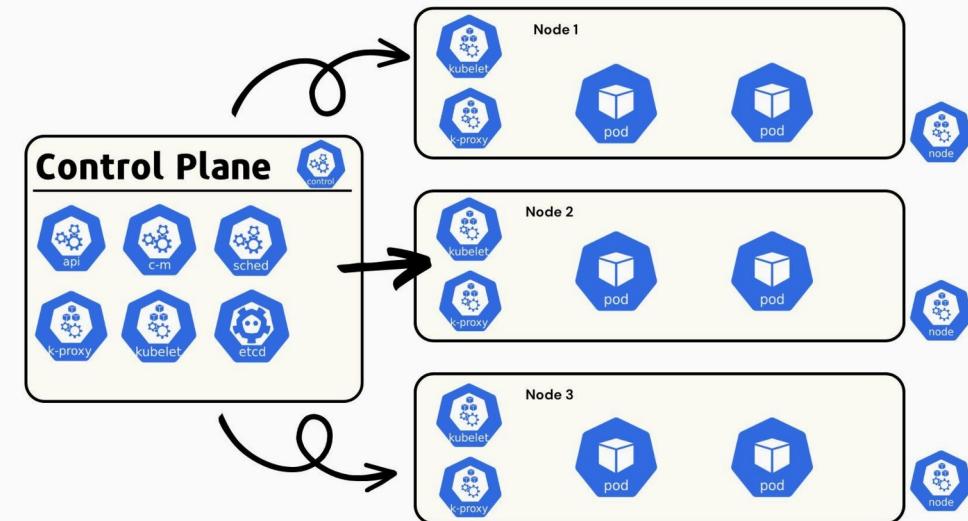


Patrones de Diseño Clave

Uso de Contenedores y Kubernetes

Los contenedores, como los ofrecidos por **Docker**, han revolucionado el diseño de sistemas distribuidos al proporcionar un entorno estandarizado y portable para ejecutar aplicaciones.

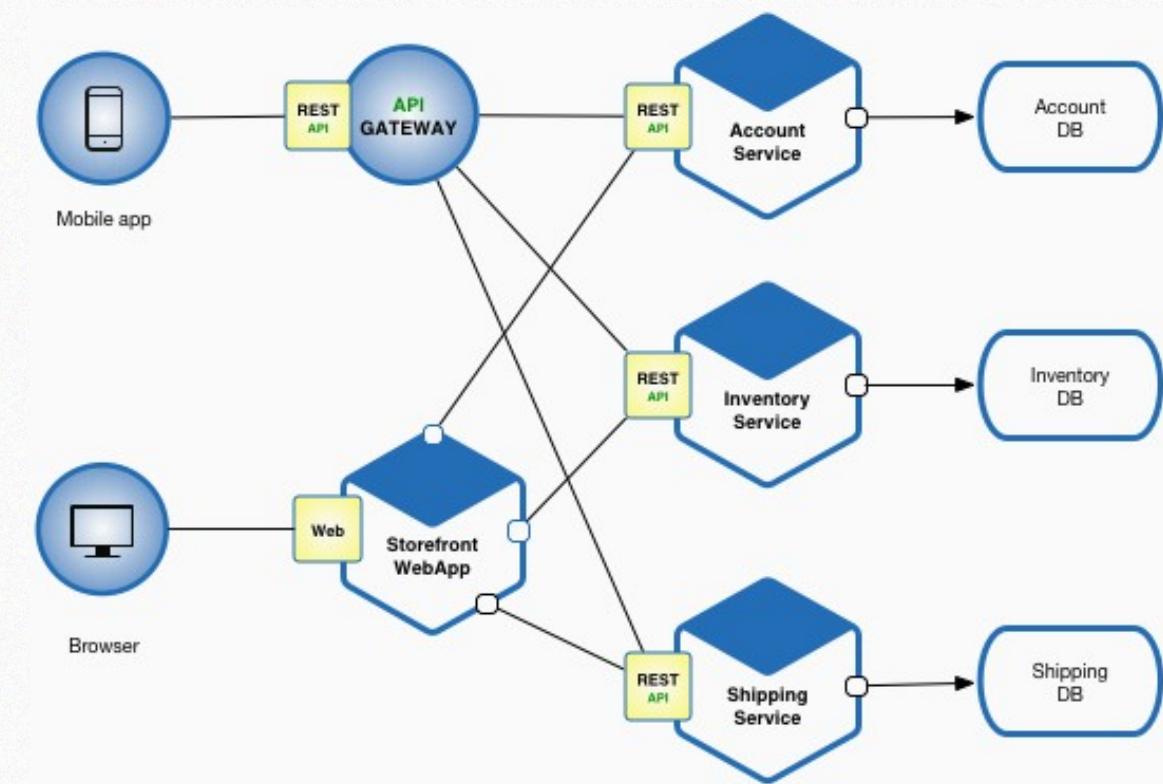
- **Ventaja:** Permiten que los desarrolladores empaqueten aplicaciones junto con sus dependencias, asegurando que se comporten de manera consistente en cualquier entorno.
- **Kubernetes:** Va un paso más allá, ofreciendo una plataforma para orquestar contenedores, manejar su escalabilidad y garantizar su alta disponibilidad.



Microservicios: Diseñando Sistemas Modulares

El enfoque de **microservicios** implica dividir una aplicación en componentes pequeños e independientes que interactúan entre sí.

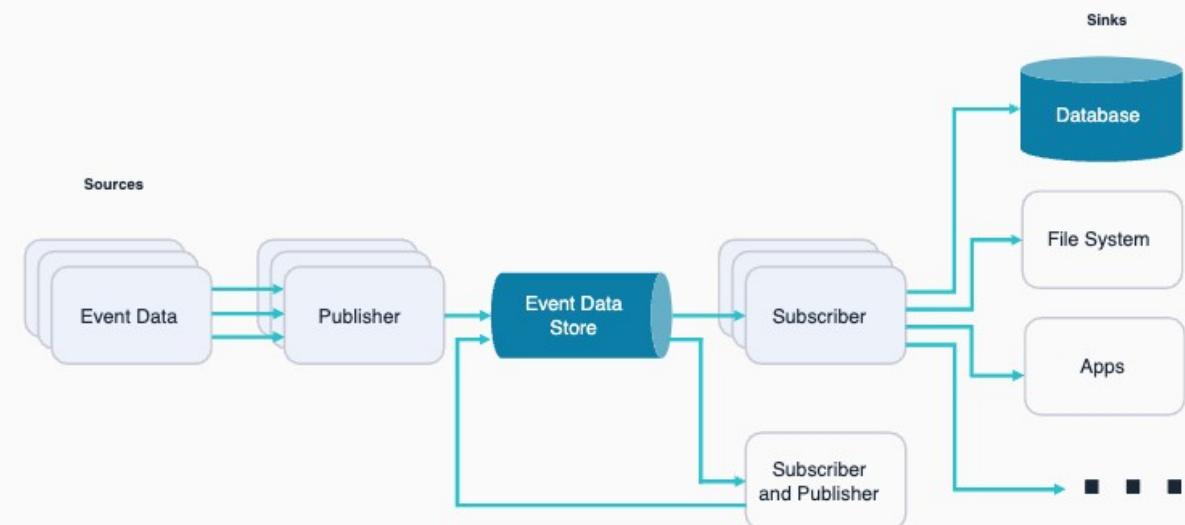
- **Ventaja:** Facilita el desarrollo y la implementación independiente de cada componente.
- **Desafío:** Incrementa la complejidad de la comunicación y requiere soluciones para problemas como la **consistencia transaccional** y la **observabilidad**.



Patrones de Comunicación: Event-driven y Mensajería

En el desarrollo de software, la comunicación es el alma del diseño. Hay dos enfoques principales:

- **RPC (Remote Procedure Call):** Llamadas sincrónicas que permiten que un servicio invoque directamente funciones en otro servicio. Ejemplo: **gRPC**.
- **Mensajería asíncrona:** Aquí los servicios se comunican mediante mensajes enviados a través de colas como **RabbitMQ** o **Kafka**.
 - Ventaja: Reduce la dependencia directa entre servicios y mejora la resiliencia.



Principios Fundamentales de los Sistemas Distribuidos

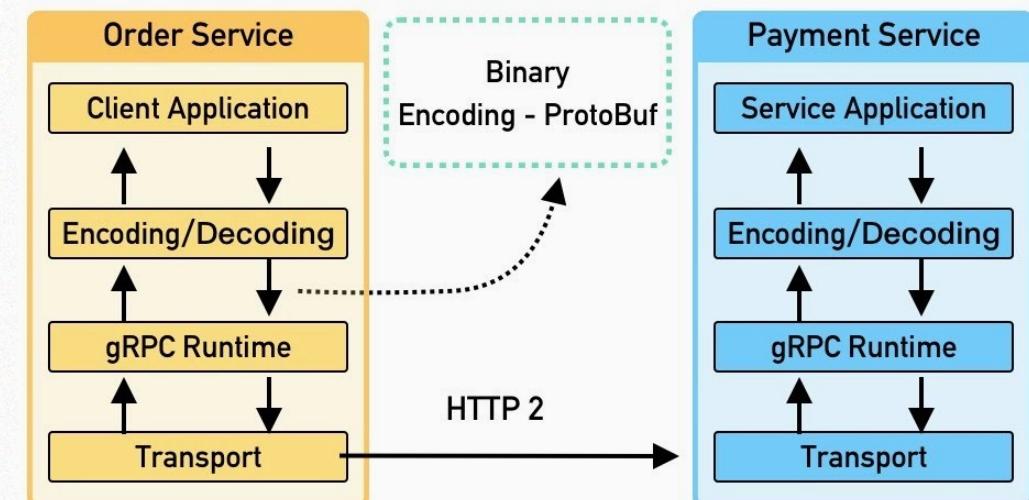
Comunicación entre procesos

En un sistema distribuido, los procesos deben comunicarse entre sí para coordinar tareas y compartir datos. Esta comunicación puede ser **sincrónica** o **asíncrona**, y se logra mediante distintos mecanismos:

- **RPC (Remote Procedure Call):** Permite que un proceso ejecute una función en otro sistema como si fuera local. Es directo, pero enfrenta problemas como la latencia de red y la tolerancia a fallos.
Ejemplo: **gRPC**.

- **Mensajería:** Es más flexible que RPC porque desacopla los procesos emisores y receptores.
Ejemplo: sistemas basados en **RabbitMQ** o **Kafka**.

Ventaja: Mayor resiliencia porque los mensajes pueden almacenarse y reenviarse en caso de fallos.



Comunicación Basada en mensajes

La comunicación más básica y fundamental en sistemas distribuidos:

Envío y recepción de mensajes:

- Los procesos se comunican enviando mensajes explícitos (usando sockets, por ejemplo).
- Ejemplo: Un cliente envía una solicitud a un servidor para acceder a un recurso, y el servidor responde con el resultado.

Protocolo de transporte:

- Puede ser basado en **TCP** (orientado a conexión, confiable) o **UDP** (sin conexión, más rápido pero menos confiable).



Comunicación basada en RPC (Remote Procedure Call)

Permite que un proceso invoque funciones de otro proceso como si fueran funciones locales.

Protocolos:

- **gRPC** (Google)
- **XML-RPC**
- **JSON-RPC**.

```
{  
  "jsonrpc": "2.0",  
  "method": "subtract",  
  "params": [  
    42,  
    23  
  ],  
  "id": 1  
}
```



Ejemplo:

solicitud:
{"method":"my_method","params":[1,2,3],"id":"my_id"}
respuesta:
{"result":"my_result","error":null,"id":"my_id"}

Ejemplo

```
from jsonrpcserver import method, serve

@method
def sumar(a, b):
    return a + b

if __name__ == "__main__":
    serve()
```

```
from jsonrpcclient import request

response = request("http://localhost:5000/", "sumar", a=5, b=3)

print(response.data.result)
```

Servidor JSON RPC

```
from jsonrpcserver import method, serve

@method
def add(a: int, b: int) -> int:
    return a + b

@method
def multiply(a: int, b: int) -> int:
    return a * b

if __name__ == "__main__":
    serve("localhost", 5000)
```

Cliente JSON RPC

```
import requests
import json

def json_rpc_request(method, params):
    payload = {
        "jsonrpc": "2.0",
        "method": method,
        "params": params,
        "id": 1
    }
    response = requests.post("http://localhost:5000", json=payload)
    return response.json()

if __name__ == "__main__":
    # Realizando una suma remota
    result_add = json_rpc_request("add", {"a": 10, "b": 20})
    print("Resultado de la suma:", result_add)

    # Realizando una multiplicación remota
    result_multiply = json_rpc_request("multiply", {"a": 5, "b": 4})
    print("Resultado de la multiplicación:", result_multiply)
```

Relojes y Sincronización

Uno de los desafíos más interesantes en sistemas distribuidos es cómo sincronizar los relojes.

En un sistema distribuido, los relojes físicos no son precisos debido a los retrasos en la red y las diferencias de hardware.

Teoría clave:

- **Relojes lógicos de Lamport:** Proponen un enfoque basado en la causalidad. En lugar de sincronizar los relojes físicos, se ordenan los eventos en función de qué evento “causó” otro.
- **Relojes vectoriales:** Una extensión más precisa que permite rastrear relaciones de causalidad entre múltiples procesos.

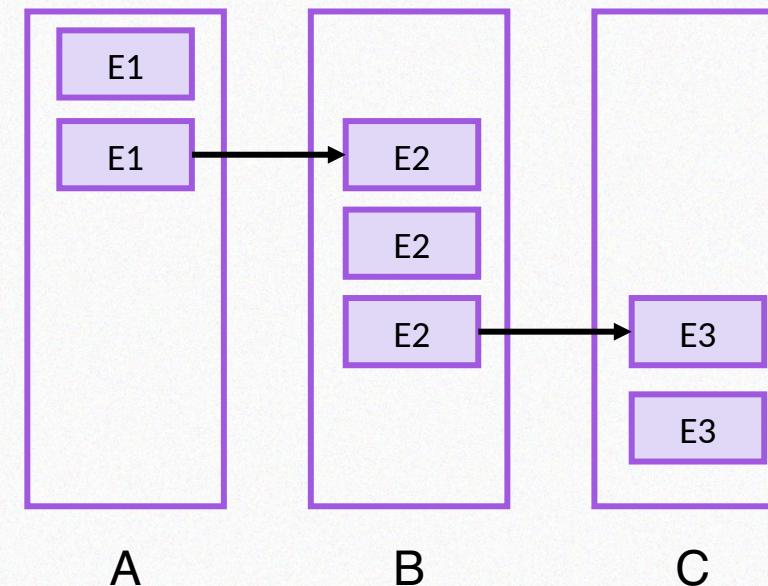


Ejemplo

Supongamos que tenemos un sistema distribuido con **tres nodos** A, B y C, y que están colaborando para procesar transacciones en un sistema financiero.

Los eventos ocurren en cada nodo y se envían mensajes entre ellos.

Queremos analizar cómo los relojes de Lamport capturan la causalidad de estos eventos.



Reloj de Lamport

$$L_A = 1$$

$$L_A = 2$$

$$\begin{aligned} L_B &= \max(L_B, L_A) \\ &\quad + 1 = 3 \\ L_B &= 4 \end{aligned}$$

$$\begin{aligned} L_C &= \max(L_C, L_B) + \\ &\quad 1 = 5 \end{aligned}$$

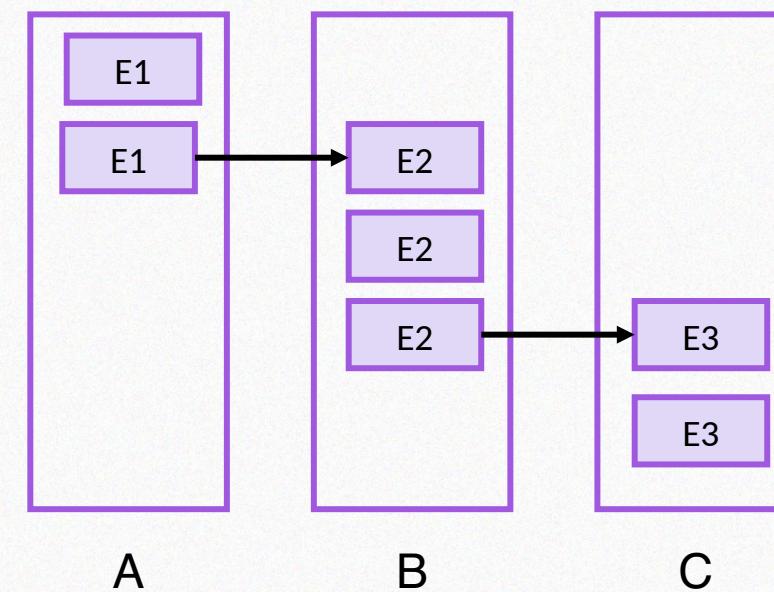
A	E1, E2	1, 2
B	E2, E3	3, 4
C	E3	5

Ejemplo

Supongamos que tenemos un sistema distribuido con **tres nodos** A, B y C, y que están colaborando para procesar transacciones en un sistema financiero.

Los eventos ocurren en cada nodo y se envían mensajes entre ellos.

Queremos analizar cómo los relojes vectoriales capturan la causalidad de estos eventos.



Reloj Vectorial

$$v_A = [1, 0, 0]$$

$$v_A = [1, 0, 0]$$

$$v_A = [1, 1, 0]$$

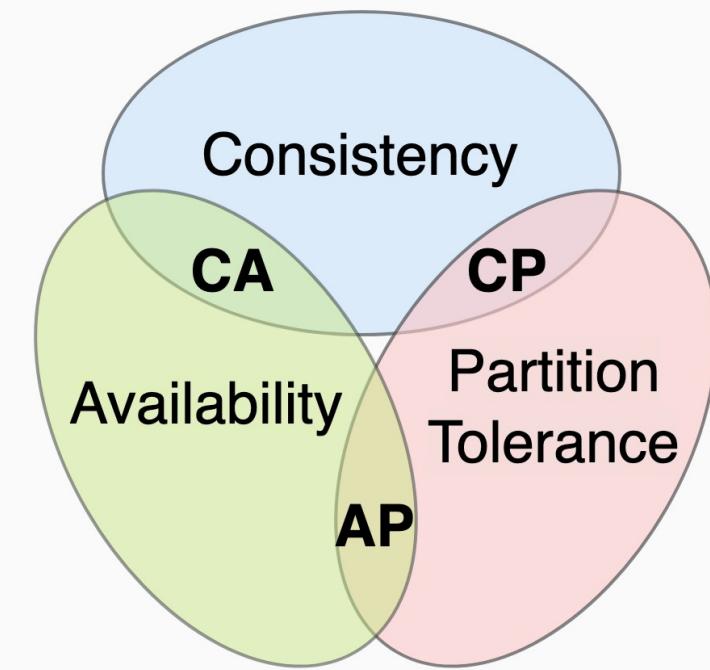
$$v_A = [2, 2, 0]$$

$$v_A = [2, 2, 1]$$

Teorema CAP

En cualquier sistema distribuido, solo puedes garantizar **dos de tres propiedades** simultáneamente:

- **Consistencia:** Todos los nodos ven los mismos datos al mismo tiempo.
- **Disponibilidad:** El sistema responde a todas las solicitudes, incluso en caso de fallos.
- **Tolerancia a particiones:** El sistema sigue funcionando incluso si hay fallos en la red.



Ejemplo práctico del Teorema CAP

Imaginemos un sistema distribuido que gestiona una base de datos bancaria con múltiples nodos.

Supongamos que un cliente realiza una transacción para transferir dinero entre dos cuentas.

1. Priorizar Consistencia y Disponibilidad (CA):

Si el sistema elige garantizar Consistencia y Disponibilidad, cualquier operación de escritura será replicada en todos los nodos antes de considerarse completada.

- **Ejemplo:** Si la transferencia ocurre, todos los nodos del sistema reflejan el saldo actualizado antes de permitir cualquier lectura. Sin embargo, si la red sufre una partición, el sistema dejará de estar disponible para evitar inconsistencias.

Ejemplo práctico del Teorema CAP

Imaginemos un sistema distribuido que gestiona una base de datos bancaria con múltiples nodos.

Supongamos que un cliente realiza una transacción para transferir dinero entre dos cuentas.

2. Priorizar Consistencia y Tolerancia a particiones (CP): Aquí, el sistema garantiza que los datos sean consistentes incluso durante una partición de la red, pero la disponibilidad puede verse comprometida.

- **Ejemplo:** Durante una partición, el sistema podría negar las solicitudes de lectura o escritura hasta que la conexión entre los nodos se restaure, asegurando que todos los datos permanezcan consistentes.

Ejemplo práctico del Teorema CAP

Imaginemos un sistema distribuido que gestiona una base de datos bancaria con múltiples nodos.

Supongamos que un cliente realiza una transacción para transferir dinero entre dos cuentas.

3. Priorizar Disponibilidad y Tolerancia a particiones (AP): Si el sistema elige Disponibilidad y Tolerancia a particiones, responderá a todas las solicitudes, incluso si algunos nodos están desconectados o hay inconsistencias temporales.

- **Ejemplo:** Durante una partición, un cliente podría ver un saldo antiguo (no actualizado), pero el sistema seguirá permitiendo lecturas y escrituras en los nodos accesibles. Esto puede resultar en inconsistencias temporales hasta que la partición se resuelva.

Relación con la realidad

Sistemas distribuidos conocidos tienden a optimizar diferentes combinaciones según su propósito:

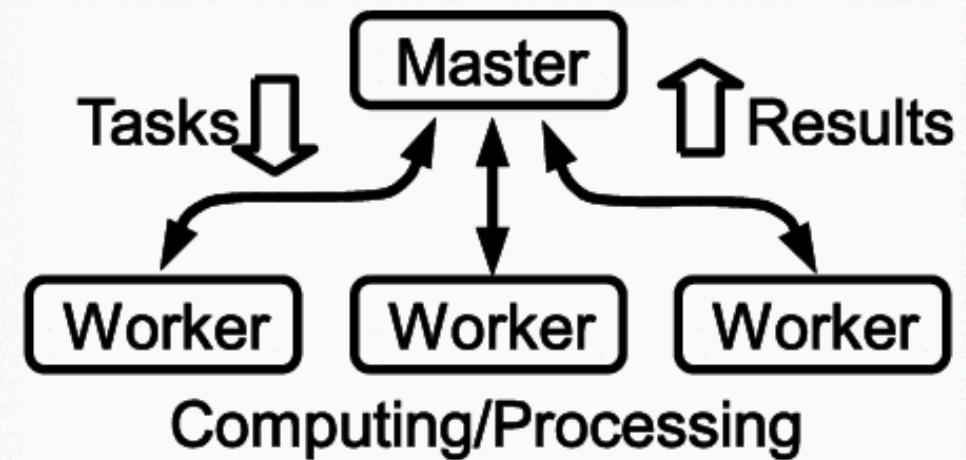
- **CP (Consistencia y Tolerancia a particiones)**: Bases de datos como MongoDB (en modo estricto) o HBase.
- **AP (Disponibilidad y Tolerancia a particiones)**: Sistemas como DynamoDB o Cassandra, donde la alta disponibilidad es crucial.
- **CA (Consistencia y Disponibilidad)**: Solo es posible en ausencia de particiones, lo que no es realista en sistemas distribuidos grandes.

Patrones de Arquitectura

Master – Worker (o Coordinator – Worker)

El patrón **Master-Worker** organiza un sistema distribuido con un nodo maestro (coordinador) que divide tareas en subtareas y las asigna a nodos trabajadores (workers).

Los trabajadores ejecutan sus tareas y devuelven los resultados al maestro, quien los combina en un resultado final.



Caso Real

Un sistema de renderizado de videos en una plataforma de edición en la nube (como Adobe Premiere en la nube). Los videos se dividen en fragmentos que se procesan en paralelo y se combinan.

Flujo:

1. El maestro divide el video en fragmentos (10 segundos cada uno).
2. Cada worker aplica efectos y codifica su fragmento.
3. El maestro combina los fragmentos en un solo archivo.

```
from multiprocessing import Pool

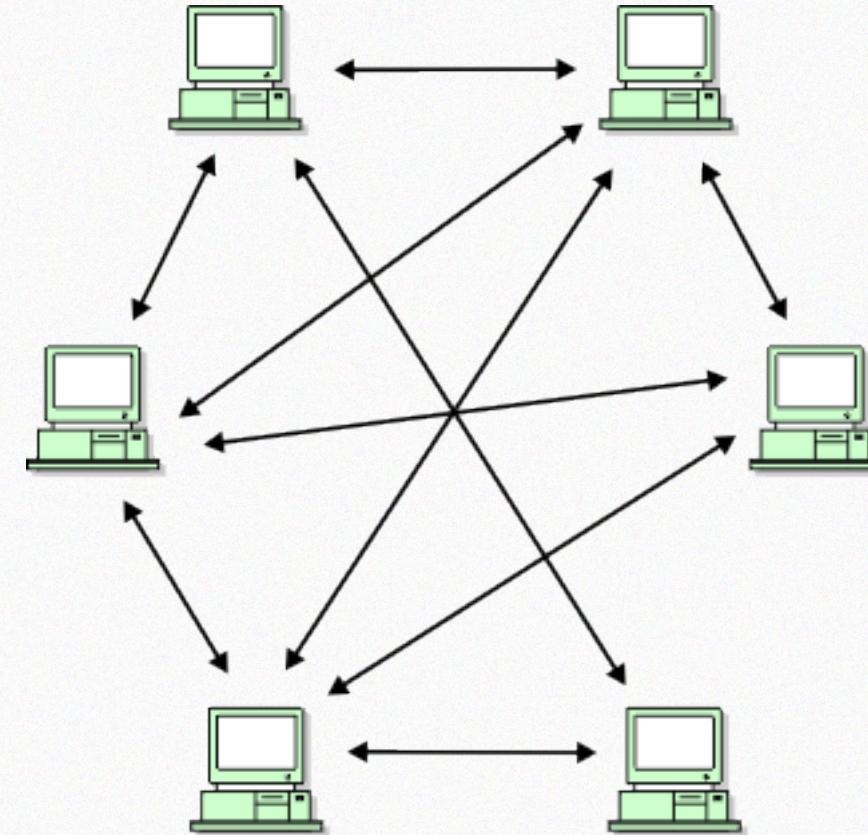
def render_frame(frame_data):
    # Simula procesamiento de un fragmento de video
    processed_frame = f"Frame {frame_data} processed"
    return processed_frame

if __name__ == "__main__":
    video_frames = list(range(1, 101)) # 100 frames
    with Pool(5) as pool: # 5 workers
        results = pool.map(render_frame, video_frames)
    print("Video procesado:", results)
```

Peer-to-Peer (P2P)

El patrón P2P organiza nodos iguales que interactúan directamente sin un servidor central. Cada nodo puede actuar como cliente o servidor.

Ejemplo: **BitTorrent**, donde los nodos comparten partes de archivos directamente entre sí.



Ejemplo

Diseñemos un sistema de mensajería instantánea donde cada nodo pueda enviar y recibir mensajes directamente entre sí, sin depender de un servidor central.

Cada nodo actúa como cliente y servidor, estableciendo conexiones directas con otros nodos para intercambiar mensajes.

```
# Función para manejar los mensajes entrantes
def handle_incoming_messages(server_socket):
    while True:
        conn, addr = server_socket.accept()
        message = conn.recv(1024).decode()
        print(f"Mensaje recibido de {addr}: {message}")
        conn.close()

# Función para enviar mensajes a otros nodos
def send_message(target_host, target_port, message):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((target_host, target_port))
    client_socket.send(message.encode())
    client_socket.close()

# Configurar el nodo como servidor
def start_node(port):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('localhost', port))
    server_socket.listen(5)
    print(f"Nuevo nodo escuchando en puerto {port}")

# Iniciar un hilo para manejar mensajes entrantes
threading.Thread(target=handle_incoming_messages, args=
```

MapReduce

Divide el procesamiento de datos en dos etapas

1. **Map:** Los nodos trabajadores procesan fragmentos de datos.
2. **Reduce:** Los resultados parciales se combinan.

```
# Coordinador principal
def distribute_and_reduce(comments, workers):
    # Dividir los comentarios en lotes
    batch_size = len(comments) // len(workers)
    batches = [comments[i:i + batch_size] for i in range(0, len(comments), batch_size)]

    # Enviar los lotes a los trabajadores
    partial_results = []
    for batch, worker_url in zip(batches, workers):
        result = json_rpc_request(worker_url, "analyze", batch)
        partial_results.append(result)

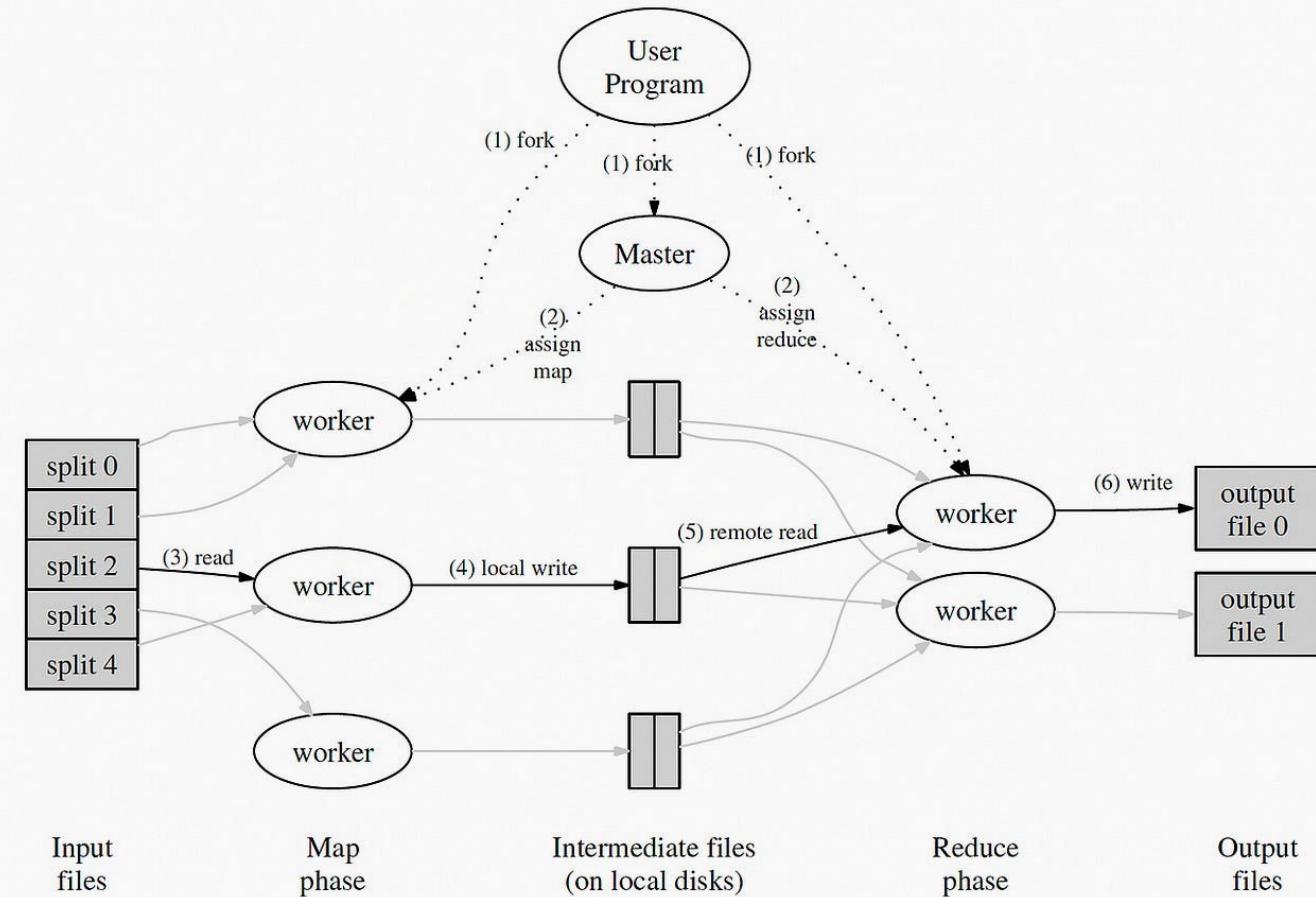
    # Reducir los resultados parciales
    final_result = {"positivo": 0, "negativo": 0}
    for partial in partial_results:
        final_result["positivo"] += partial["positivo"]
        final_result["negativo"] += partial["negativo"]

    return final_result
```

MapReduce

Ampliamente utilizado por Google en sus sistemas de indexación, sigue este pa-

1. **Map Phase:** Divide los datos en fragmentos y distribuye la tarea a los workers para procesarlos.
2. **Reduce Phase:** Los workers devuelven resultados parciales que se combinan en un único resultado.



Actor Model

El modelo de actor define entidades independientes que manejan estado y se comunican mediante mensajes asíncronos.

1. **Actor:** La unidad básica que realiza tareas, mantiene estado y responde a mensajes. Cada actor puede:

- Recibir y procesar un mensaje.
- Crear nuevos actores.
- Enviar mensajes a otros actores (identificados por direcciones o referencias).

2. **Mensajes:** Los mensajes son **inmutables** y se envían de manera asíncrona.

3. **Sistema de Mensajería:** Una cola de mensajes (inbox) asociada a cada actor garantiza que los mensajes lleguen y se procesen uno por uno.



Implementación práctica

El modelo de actor define entidades independientes que manejan estado y se comunican mediante mensajes asíncronos.

El patrón Actor es popularmente implementado en:

- **Akka (Java/Scala):** Framework que usa el patrón Actor para sistemas concurrentes y distribuidos.
- **Erlang/Elixir:** Lenguajes diseñados desde cero con el modelo de actores para soportar alta concurrencia y resiliencia.
- **Ray:** Biblioteca Python para computación distribuida basada en actores.



Ejemplo:

Este ejemplo modela un sistema básico de gestión de pedidos en un restaurante.

```
public class OrderActor extends UntypedAbstractActor {  
    private final ActorRef kitchen;  
    private final ActorRef notifier;  
  
    public OrderActor(ActorRef kitchen, ActorRef notifier) {  
        this.kitchen = kitchen;  
        this.notifier = notifier;  
    }  
  
    @Override  
    public void onReceive(Object message) {  
        if (message instanceof String) {  
            String order = (String) message;  
            System.out.println("OrderActor: Pedido recibido: " + order);  
  
            // Enviar el pedido a la cocina  
            kitchen.tell(order, getSelf());  
        } else if (message instanceof Boolean && (Boolean) message) {  
            System.out.println("OrderActor: ¡Pedido listo! Gracias por su  
paciencia.");  
        } else {  
            unhandled(message);  
        }  
    }  
}
```

Consistencia y Replicación

¿Qué es la consistencia en un sistema distribuido?

La consistencia se refiere a la **garantía de que los datos distribuidos entre múltiples nodos reflejan la misma información en un momento dado**. Es uno de los mayores retos en sistemas distribuidos, ya que los nodos pueden estar en diferentes ubicaciones, con redes sujetas a latencias, fallos o particiones.



Imagina una base de datos replicada en tres servidores ubicados en diferentes países (Londres, Nueva York y Tokio). Si un cliente realiza una actualización en Tokio, **¿cómo aseguramos que los clientes en Londres y Nueva York vean ese cambio sin inconsistencias?**

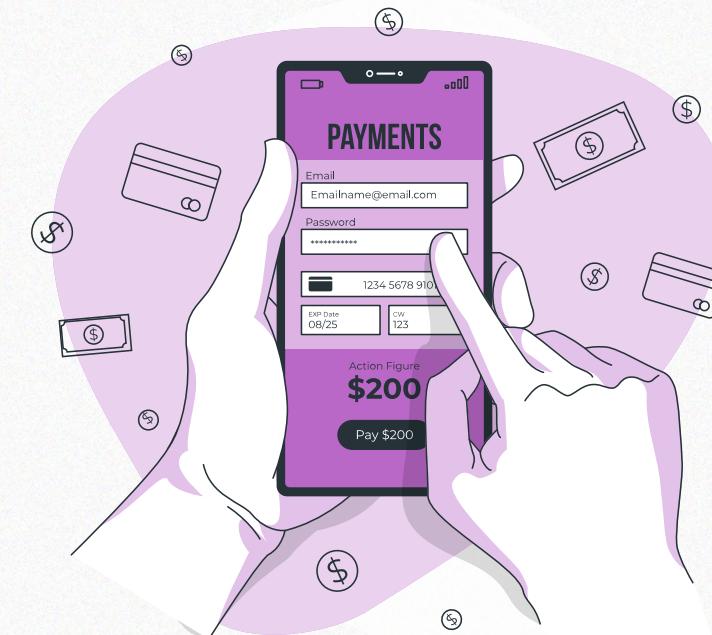
Tipos de consistencia

Consistencia fuerte

Garantiza que todos los nodos siempre muestren el mismo estado del sistema después de cualquier operación de escritura. En otras palabras, una lectura en cualquier nodo devolverá el dato más reciente.

Supongamos que estás utilizando un sistema bancario. Si transfieres \$100 de tu cuenta a la cuenta de otra persona, esperas que:

- **Tu saldo refleje inmediatamente la deducción.**
- **El saldo del destinatario aumente de inmediato.**



Tipos de consistencia

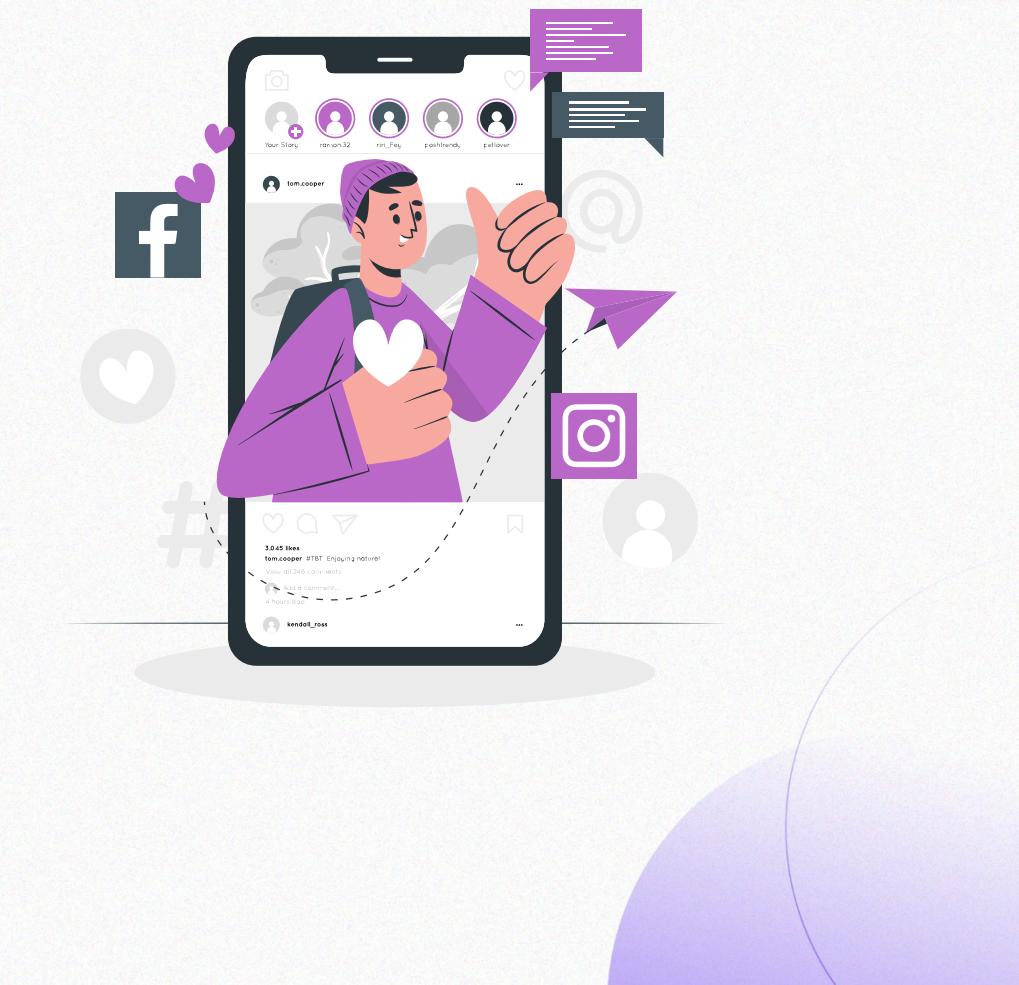
Consistencia eventual

Garantiza que, en ausencia de nuevas escrituras, todos los nodos convergerán eventualmente al mismo estado. No asegura que una lectura siempre devuelva el dato más reciente.

Redes sociales como Twitter o Instagram

- Publicas una foto en Instagram.
- Es posible que un amigo en Nueva York vea la publicación inmediatamente, pero otro amigo en Tokio tarde unos segundos en verla debido a la sincronización entre los servidores.

Aunque inicialmente hay discrepancias, todos los nodos eventualmente tendrán la publicación.



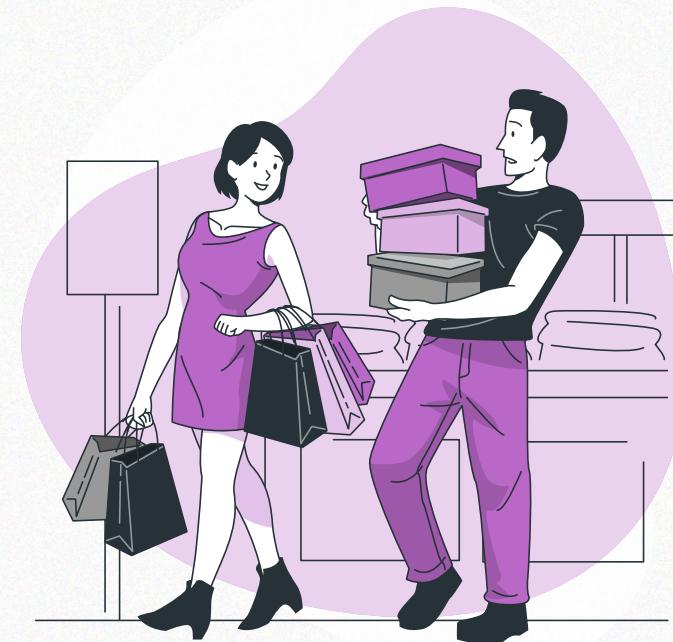
Tipos de consistencia

Consistencia relajada

Es un compromiso entre consistencia fuerte y eventual. Permite ciertas inconsistencias bajo condiciones controladas para lograr mejor rendimiento o disponibilidad.

Supongamos un sistema de inventario para una tienda en línea:

- Dos clientes intentan comprar el mismo artículo al mismo tiempo.
- En un sistema con consistencia relajada, ambos clientes pueden realizar la compra porque los nodos aún no han sincronizado el cambio en el inventario.
- Más tarde, el sistema resuelve el conflicto, notificando a uno de los clientes que el artículo ya no está disponible.



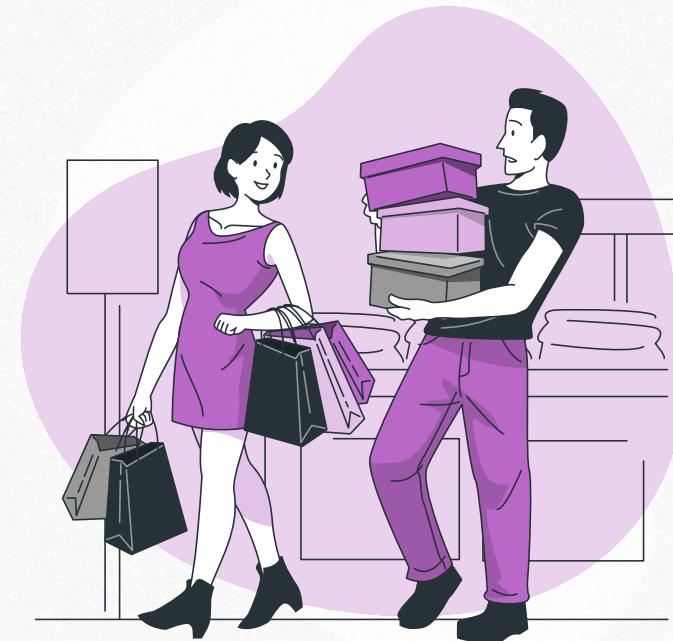
Modelos de Consistencia

Lectura/Escritura Primaria

En este modelo, un nodo principal maneja todas las escrituras y coordina la replicación hacia los nodos secundarios, que se encargan principalmente de las lecturas.

Un sistema de base de datos para una tienda en línea:

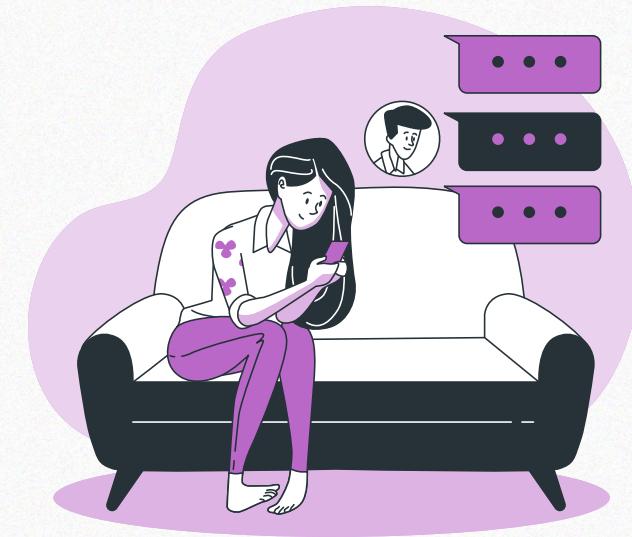
- Todas las actualizaciones (como cambios en el inventario) son gestionadas por un nodo primario.
- Los nodos secundarios replican los datos del primario y manejan las solicitudes de lectura para reducir la carga en el nodo principal.



Replicación en Sistemas Distribuidos

La replicación es una técnica fundamental en sistemas distribuidos que consiste en **mantener múltiples copias de los datos en diferentes nodos o ubicaciones**.

Esto se realiza para mejorar la disponibilidad, el rendimiento, la tolerancia a fallos y la escalabilidad del sistema.

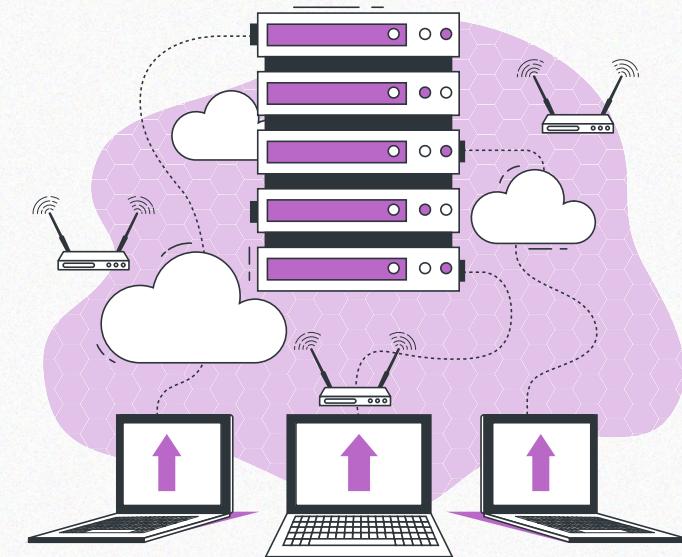


Imagina que tienes una aplicación de mensajería global como WhatsApp. Los usuarios envían mensajes desde diferentes partes del mundo, y el sistema necesita almacenar y procesar esos datos rápidamente.

Si los datos se almacenan en un solo servidor, los usuarios distantes experimentarán alta latencia y, si el servidor falla, todo el sistema colapsará.

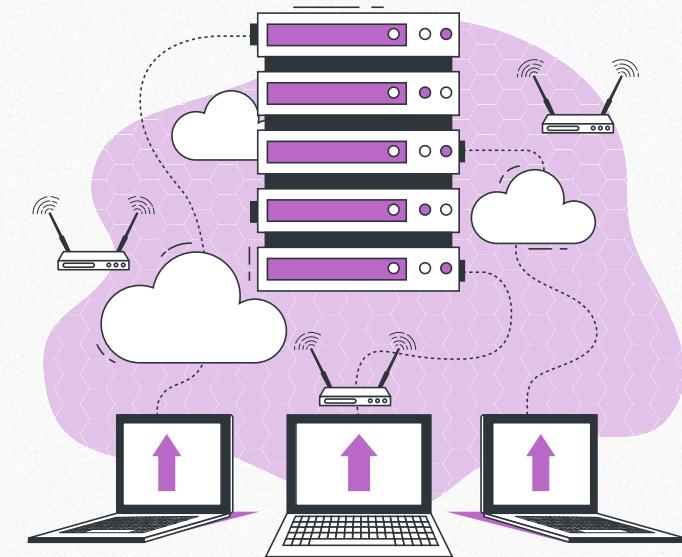
Objetivos de la replicación

- 1. Disponibilidad:** Si un nodo falla o no está disponible, otros nodos replicados pueden asumir su carga.
- 2. Rendimiento:** Al distribuir las copias, las lecturas pueden ser atendidas localmente, reduciendo la latencia y equilibrando la carga.
- 3. Tolerancia a fallos:** La pérdida de datos se minimiza porque siempre hay copias redundantes disponibles.
- 4. Escalabilidad:** Se pueden añadir más nodos replicados para manejar mayores volúmenes de tráfico.



Objetivos de la replicación

- 1. Disponibilidad:** Si un nodo falla o no está disponible, otros nodos replicados pueden asumir su carga.
- 2. Rendimiento:** Al distribuir las copias, las lecturas pueden ser atendidas localmente, reduciendo la latencia y equilibrando la carga.
- 3. Tolerancia a fallos:** La pérdida de datos se minimiza porque siempre hay copias redundantes disponibles.
- 4. Escalabilidad:** Se pueden añadir más nodos replicados para manejar mayores volúmenes de tráfico.



Replicación Síncrona

En la replicación síncrona, **los datos se actualizan en todos los nodos replicados antes de confirmar la operación al cliente**. Esto asegura que todas las copias estén sincronizadas en todo momento.

Funcionamiento:

1. Un cliente realiza una operación de escritura (por ejemplo, actualizar un registro en la base de datos).
2. El sistema distribuye esta operación a todos los nodos replicados.
3. Hasta que todos los nodos confirmen la escritura, la operación no se considera completada.



Google Spanner

- Usa replicación síncrona entre múltiples centros de datos (data centers).
- Cada escritura espera a que una mayoría (quorum) de réplicas en distintas regiones confirme haber recibido los datos.

Cuando tú ves una canción nueva en tu cuenta de Spotify o haces una compra en Twitch, hay miles de servidores asegurándose de que esa información se vea igual sin importar desde dónde accedas. Google Spanner permite que eso pase **sin errores ni retrasos**, y por eso estas empresas lo usan.

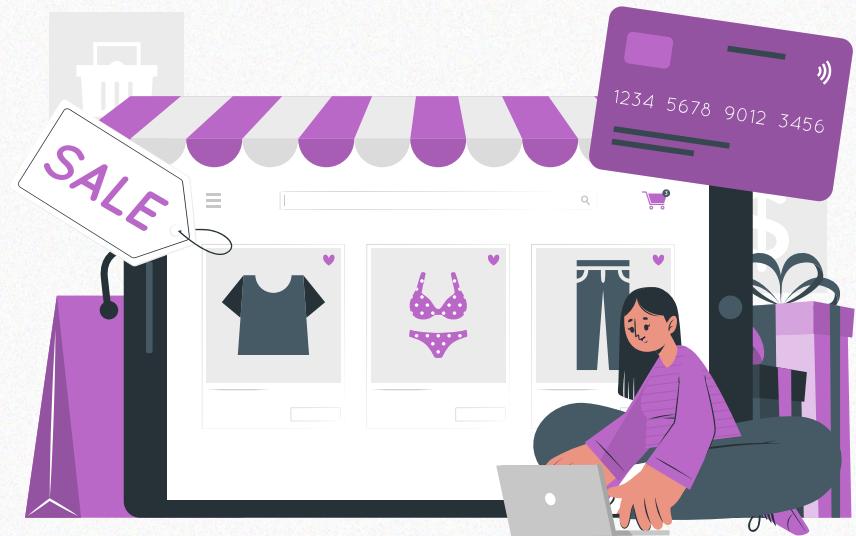
Google
Cloud
Spanner



Ejemplo

Supongamos un sistema de control de inventario global para una cadena de supermercados.

- Si un cliente compra el último artículo en una tienda en Lima, este cambio debe reflejarse inmediatamente en las bases de datos replicadas para evitar que otro cliente en una tienda de Cusco intente comprar ese mismo artículo.
- Esto es posible porque la replicación síncrona asegura consistencia fuerte: todos los nodos muestran el mismo estado actualizado.

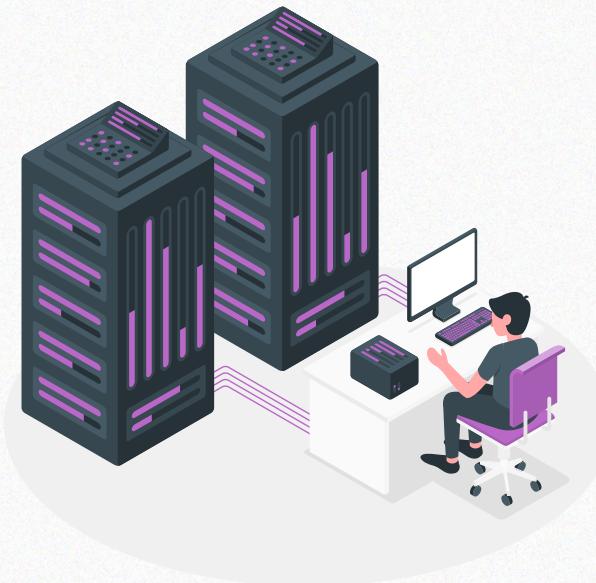


Replicación Asíncrona

En la replicación asíncrona, **el nodo principal confirma la operación de escritura inmediatamente y luego propaga los cambios a los nodos replicados en segundo plano.**

Funcionamiento:

1. Un cliente realiza una operación de escritura en el nodo principal.
2. El nodo principal responde al cliente indicando que la operación se ha completado.
3. Los datos se replican a otros nodos de forma diferida.

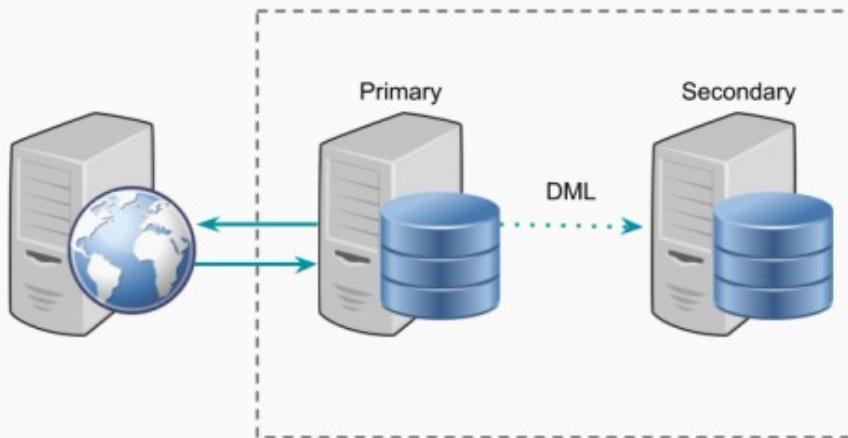


MySQL en modo replica

Si usas *replicación asíncrona*, los datos se escriben primero en el nodo principal (master), y los esclavos (replicas) lo reciben después. Esto permite escalar lecturas, pero si el master cae antes de replicar, se pierde esa info.

Supongamos una tienda online:

- Un usuario compra zapatillas → se hace un INSERT en la tabla orders.
- El INSERT se guarda **de inmediato en el master**.
- El master responde: “OK, orden registrada”
- Mientras tanto, el cambio es enviado a las réplicas... pero esto puede tardar 1 segundo o varios (según carga/red).



Ejemplo:

Imagina un sistema de correo electrónico:

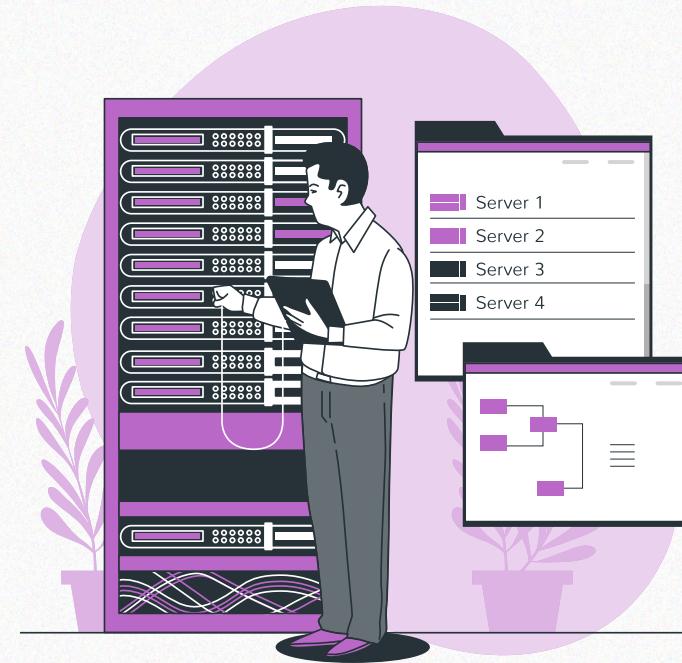
- Cuando envías un correo, el servidor principal almacena inmediatamente el mensaje y te confirma que fue enviado.
- Más tarde, el mensaje se replica en otros servidores para garantizar redundancia y permitir acceso desde múltiples dispositivos.



Replicación Híbrida

Algunos sistemas implementan una combinación de replicación síncrona y asíncrona, dependiendo de los datos o la criticidad de las operaciones.

- Las escrituras críticas (como transacciones bancarias) usan replicación síncrona.
- Las escrituras no críticas (como logs de auditoría) usan replicación asíncrona.

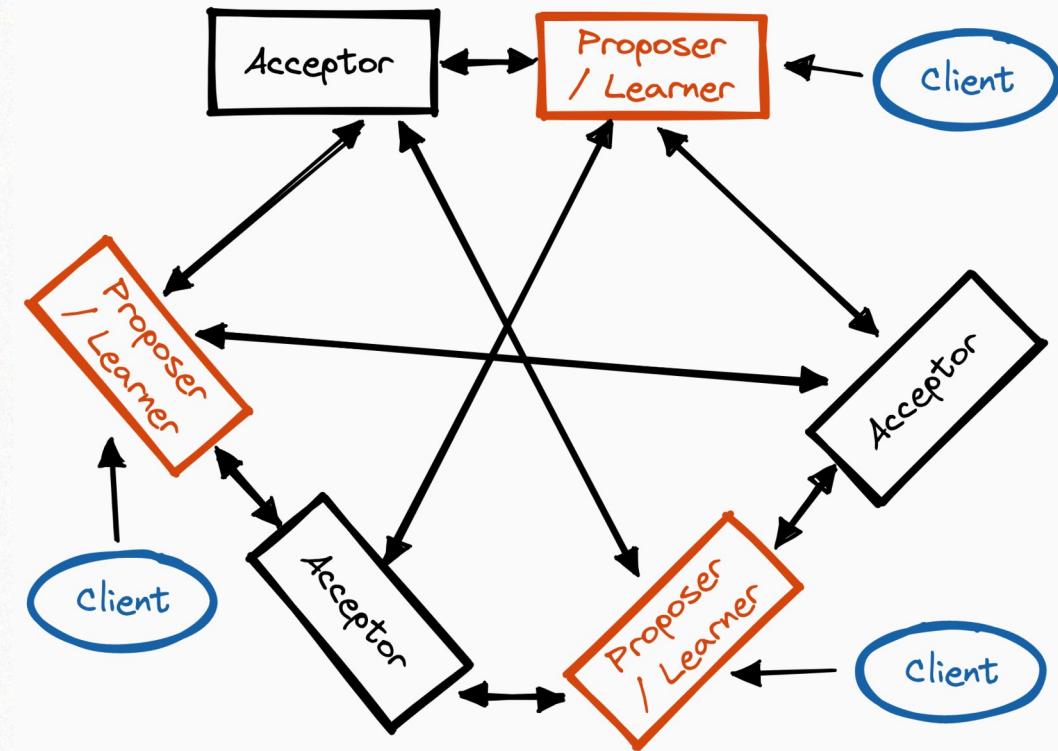


Protocolos de replicación

La replicación en sistemas distribuidos requiere protocolos para coordinar cómo los nodos replican datos de manera eficiente y consistente.

Paxos

- Paxos es un protocolo de consenso que asegura que múltiples nodos lleguen a un acuerdo sobre un valor incluso en presencia de fallos.
- Es muy robusto, pero complejo de implementar, por lo que suele utilizarse en sistemas que requieren alta tolerancia a fallos, como bases de datos distribuidas.



Elementos de Paxos

Paxos divide el proceso de consenso en tres roles principales:

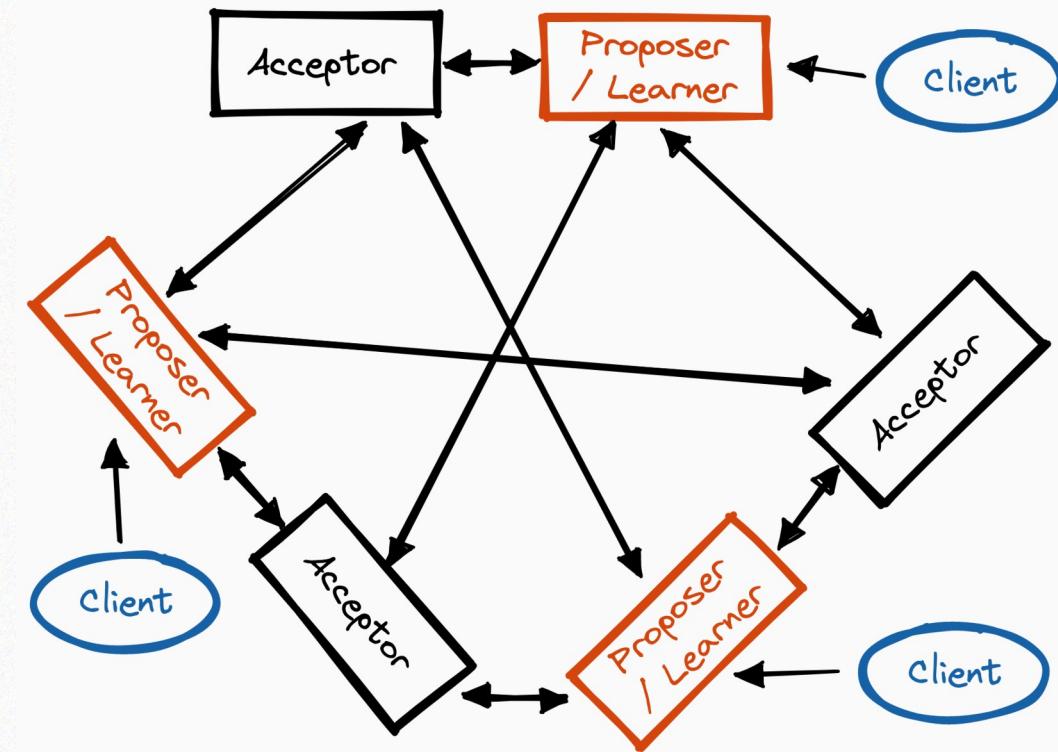
1. Proponentes (Proposers): Son los nodos que intentan proponer un valor (por ejemplo, “Comamos sushi”).

- Necesitan que una **mayoría** de los nodos acepten su propuesta para que se convierta en la decisión final.

2. Aceptadores (Acceptors): Son los nodos encargados de decidir si aceptan o no la propuesta de un Proposer.

- Para que un valor se convierta en la decisión final, debe ser aceptado por una mayoría de Acceptors.

3. Aprendices (Learners): Son los nodos que simplemente observan el valor final acordado y lo utilizan.



Fases del protocolo Paxos

Paxos se ejecuta en dos fases principales:

Fase 1: Preparación

1. Un **Proposer** elige un número de propuesta único (n) y lo envía a los **Acceptors** con el mensaje **PREPARE(n)**.
2. Cada **Acceptor**:
 - Si no ha aceptado otra propuesta, responde con **PROMISE(n)**, prometiendo que no aceptará propuestas con números menores a n .
 - Si ya ha aceptado otra propuesta con un número menor ($m < n$), envía la propuesta más reciente que ha aceptado.

Fase 2: Aceptación

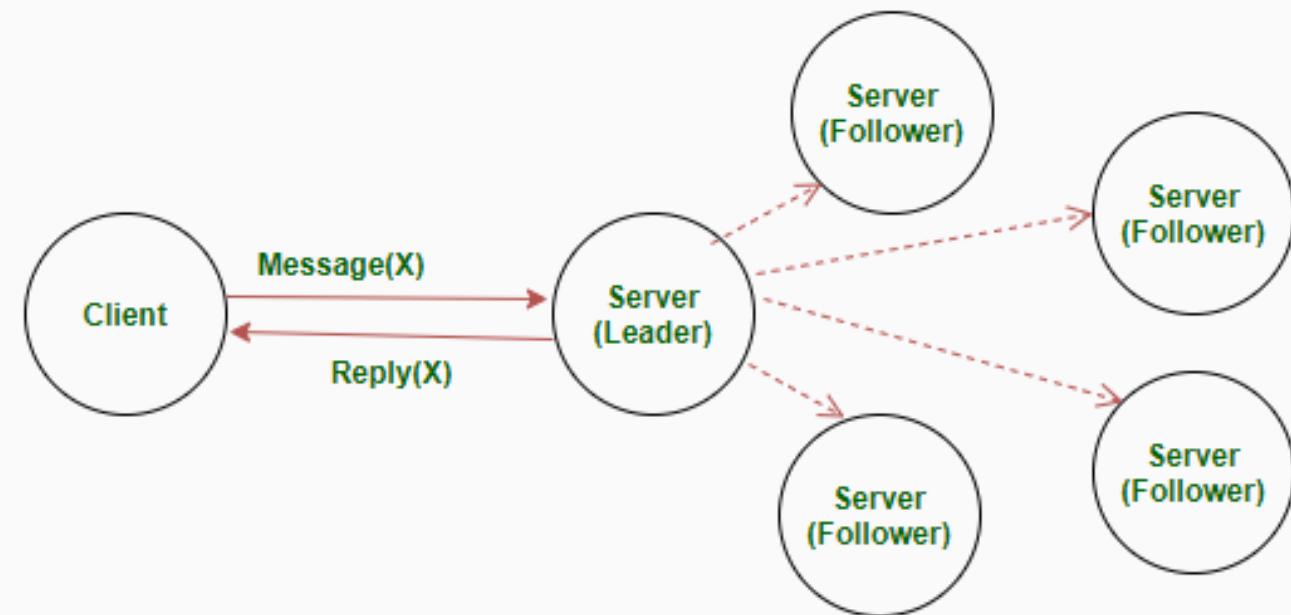
3. Si el Proposer recibe promesas (**PROMISE(n)**) de una mayoría de Acceptors, envía un mensaje **ACCEPT(n , value)** con su valor propuesto.
4. Cada **Acceptor**:
 - Si aún no ha prometido otro número mayor, acepta la propuesta y la almacena como su valor definitivo.
 - Si otro Proposer ha enviado una propuesta con un número mayor, la rechaza.

Protocolos de replicación

La replicación en sistemas distribuidos requiere protocolos para coordinar cómo los nodos replican datos de manera eficiente y consistente.

Raft

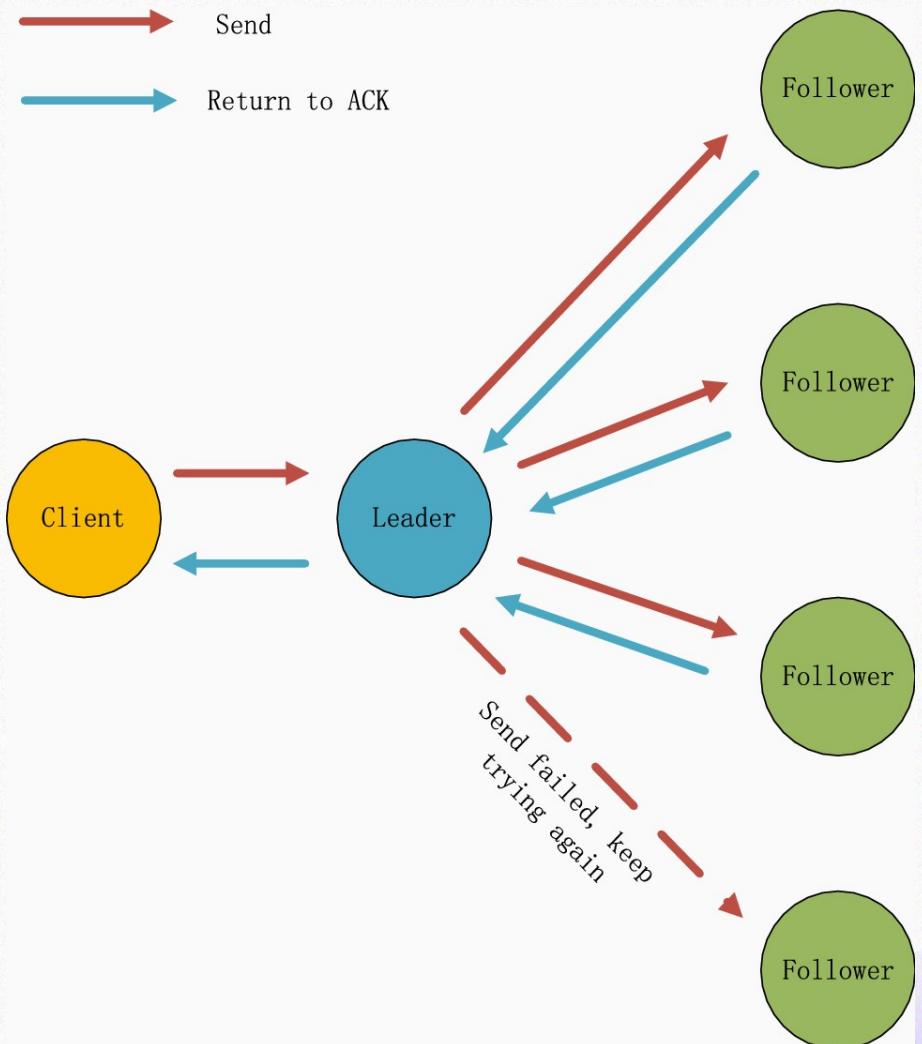
- Raft es más simple y comprensible que Paxos, pero igual de poderoso.
- Funciona eligiendo un líder entre los nodos. Este líder maneja todas las escrituras y se asegura de replicarlas a los demás nodos.



Elementos de Raft

Raft introduce tres roles en un sistema distribuido:

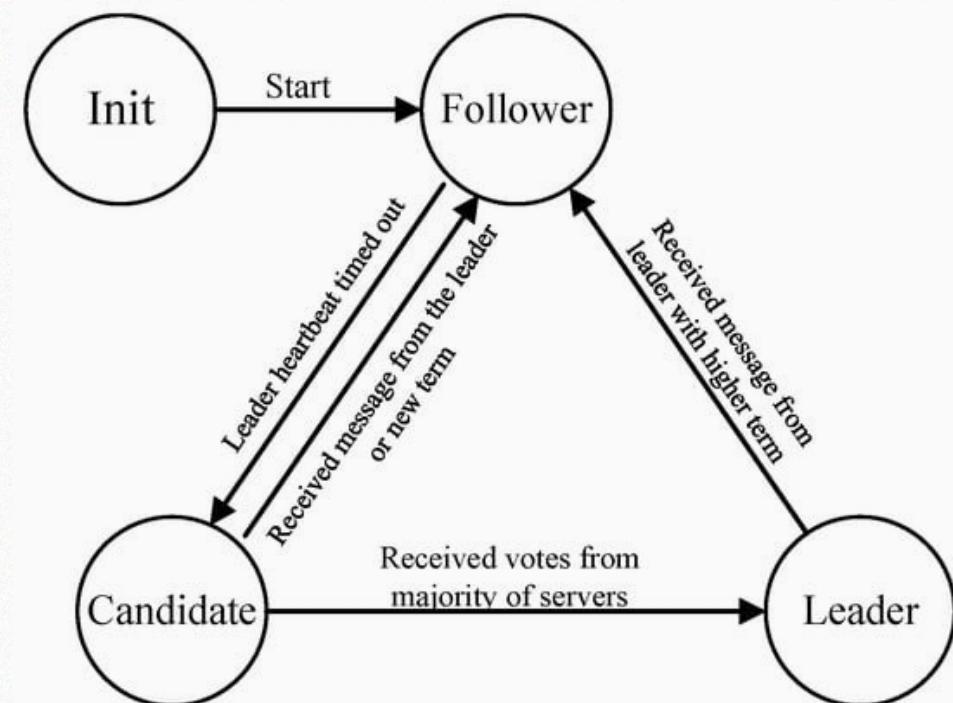
1. **Líder (Leader)**: Es el nodo que toma decisiones y propaga cambios a los demás nodos.
 - Solo hay **un líder a la vez**.
2. **Seguidores (Followers)**: Son nodos pasivos que aceptan las decisiones del líder.
 - Solo replican los datos que el líder les envía.
3. **Candidatos (Candidates)**: Si el líder desaparece, un seguidor se convierte en candidato y **solicita votos** para convertirse en el nuevo líder.



Ejemplo práctico de Raft

Imagina que tienes un **sistema de almacenamiento distribuido**:

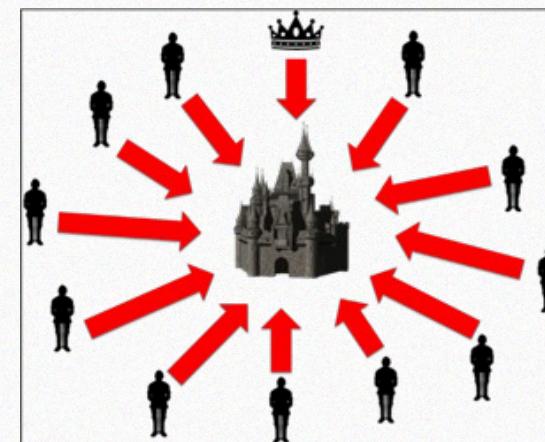
- Un usuario sube un archivo.
- El **líder** lo almacena y ordena a los seguidores que lo repliquen.
- Solo cuando la mayoría de los seguidores confirma que tienen el archivo, el líder lo da por escrito.
- Si el líder falla, se elige otro, asegurando que los datos sigan consistentes.



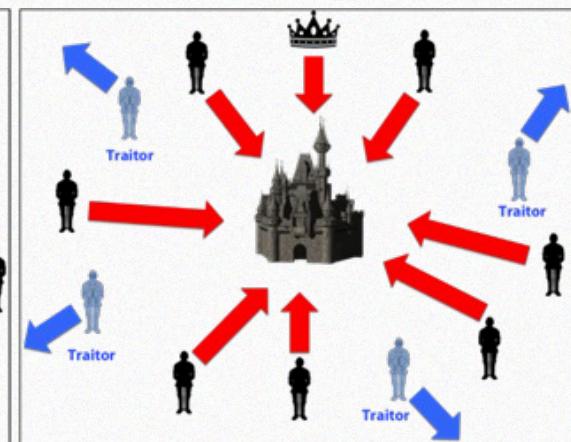
Consenso Tolerante a Fallos Bizantinos (BFT)

La tolerancia a fallos bizantinos (BFT, por sus siglas en inglés: *Byzantine Fault Tolerance*) se refiere a la capacidad de un sistema distribuido para seguir funcionando correctamente incluso si algunos de los nodos actúan de manera incorrecta, maliciosa o envían mensajes contradictorios al resto de los nodos.

Este tipo de fallos es uno de los más desafiantes en sistemas distribuidos porque no solo abarca fallos comunes, como nodos que dejan de responder, sino también nodos que intentan sabotear activamente el sistema.



Coordinated Attack Leading to Victory

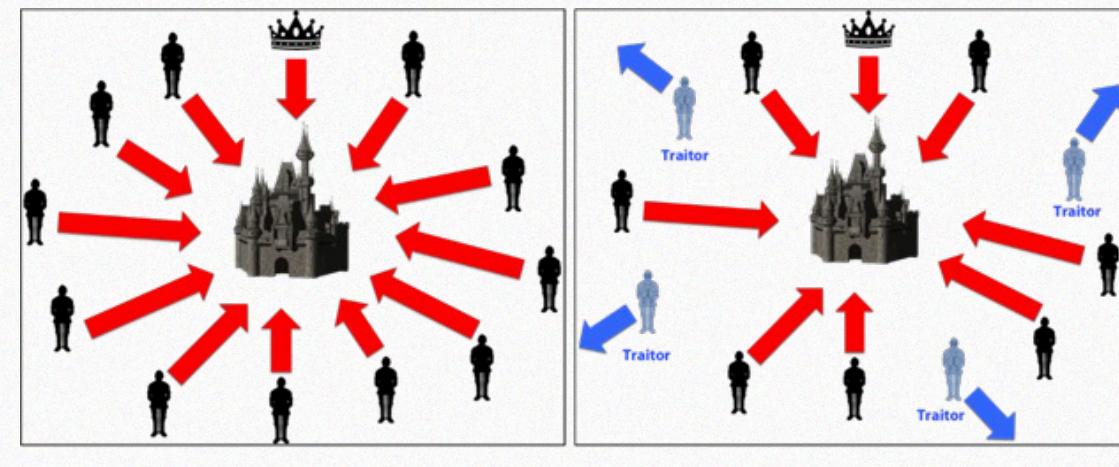


Uncoordinated Attack Leading to Defeat

¿Por qué se llaman fallos bizantinos?

El término proviene del “problema de los generales bizantinos”, propuesto por Leslie Lamport en 1982. Este problema plantea una situación en la que un grupo de generales bizantinos debe coordinarse para atacar o retirarse. Sin embargo, algunos de ellos pueden ser traidores que envían información falsa o contradictoria.

El desafío: Los generales leales deben llegar a un acuerdo sobre un plan común, incluso si los traidores intentan sabotearlo.



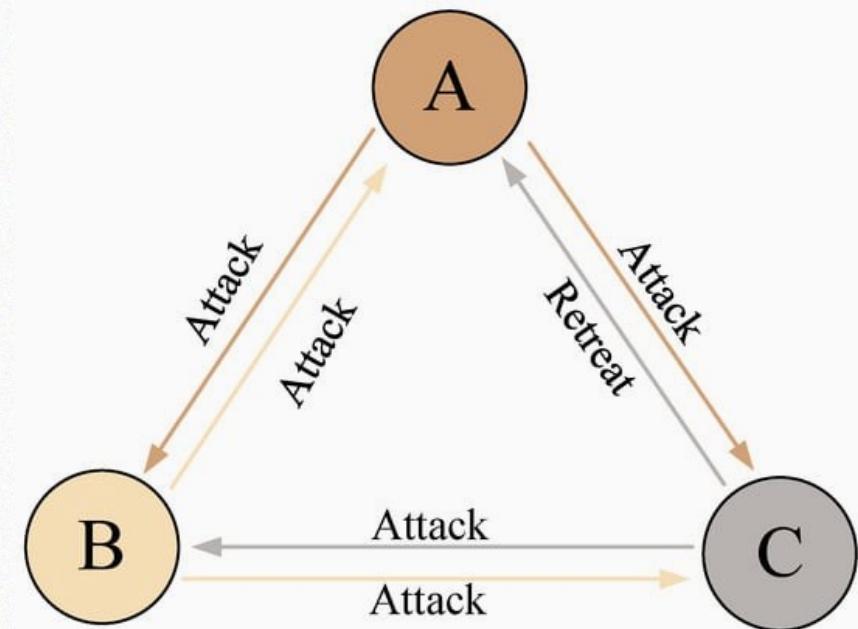
Contexto de BFT en sistemas distribuidos

En un sistema distribuido, los fallos bizantinos incluyen:

1. **Nodos maliciosos:** Nodos que deliberadamente envían información incorrecta.
2. **Fallos arbitrarios:** Nodos que operan de forma errática debido a errores de hardware o software.
3. **Mensajes falsificados:** Actores externos que intentan atacar el sistema (e.g., ataques de red o hacking).

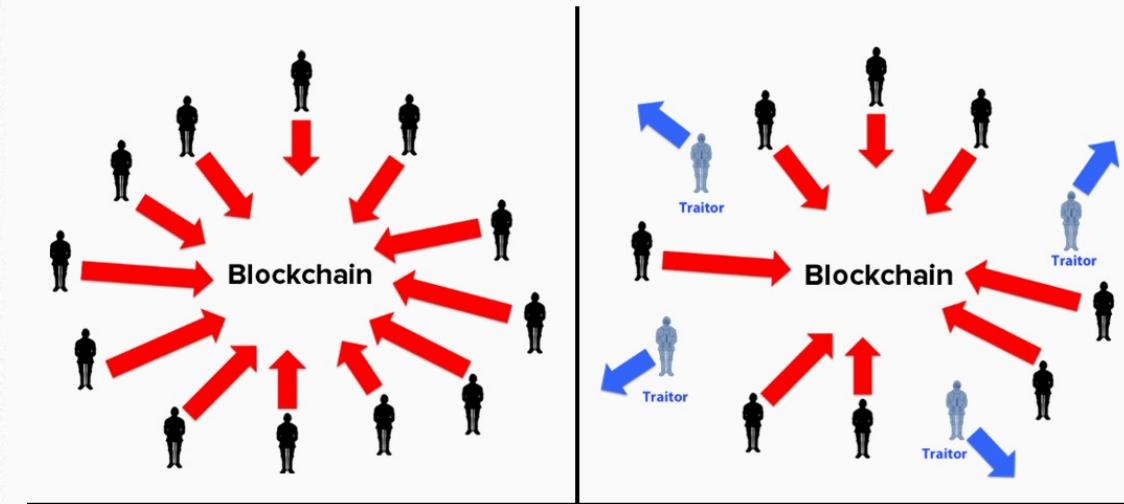
El objetivo de los algoritmos BFT es garantizar que:

- Todos los nodos correctos (honestos) lleguen a un consenso válido.
- El sistema pueda operar incluso si un número limitado de nodos es bizantino.



Características clave de los algoritmos BFT

- Tolerancia a un número limitado de nodos fallidos:** Un sistema BFT puede tolerar hasta f fallos bizantinos si tiene al menos $3f + 1$ nodos en total. Esto significa que, para soportar un nodo malicioso, el sistema debe tener al menos 4 nodos.
- Garantía de consenso:** Los nodos honestos llegarán al mismo acuerdo, sin importar el comportamiento de los nodos maliciosos.
- Eficiencia:** Aunque los algoritmos BFT son más costosos que los protocolos como Paxos o Raft, son esenciales en escenarios donde la seguridad es crítica.



All validators agree on the state of the blockchain

Only some validators agree on the state of the blockchain

Características clave de los algoritmos BFT

1. Request (cliente → primary)

- El cliente envía una solicitud al nodo principal:
“Transferir \$10 de A a B”

2. Pre-prepare (primary → backups)

- El nodo principal le dice a los demás:
“Vamos a procesar esta operación como la #42”

3. Prepare (backups → todos)

- Cada nodo backup responde a **todos los demás nodos** (no solo al primary), diciendo:

“Estoy de acuerdo en que la operación #42 es válida”

Después de recibir **al menos $2f + 1$ prepares** (incluido el suyo), un nodo sabe que **la mayoría honesta está de acuerdo**.

Características clave de los algoritmos BFT

4. Commit (todos → todos)

- Cada nodo envía un mensaje de **commit** a los demás:

“Estoy listo para ejecutar la operación #42”

Cuando un nodo recibe **$2f + 1$ commits**, ejecuta la operación.

5. Reply (réplicas → cliente)

- Cada nodo responde al cliente:

“La transferencia fue ejecutada”

El cliente espera **$f + 1$ respuestas coincidentes** para estar seguro de que la operación fue ejecutada correctamente.

Coordinación y Algoritmos Distribuidos

¿Por qué es necesaria la coordinación en sistemas distribuidos?

En un sistema centralizado, un único nodo toma decisiones y controla todas las operaciones. Sin embargo, en un sistema distribuido:

- No existe un único nodo de control.
- Los nodos pueden fallar en cualquier momento.
- La comunicación entre nodos puede ser lenta o poco confiable.
- Es posible que diferentes nodos tengan información parcial o desactualizada.



Problemas clásicos de Coordinación en Sistemas Distribuidos

Los algoritmos de coordinación buscan resolver varios problemas clave en sistemas distribuidos. A continuación, explicaremos los más importantes.

Problema de Elección de Líder

En muchos sistemas distribuidos, es necesario que un nodo asuma la función de **líder** para coordinar operaciones. El líder puede encargarse de asignar tareas, gestionar replicación o tomar decisiones en nombre de los demás nodos.



Problemas clásicos de Coordinación en Sistemas Distribuidos

Ejemplo práctico

Imagina una red de servidores que manejan peticiones de usuarios. Si todos los servidores intentan responder simultáneamente sin coordinación, pueden ocurrir colisiones y datos inconsistentes. Por eso, uno de los servidores debe ser designado como **líder**, y los demás deben seguir sus instrucciones.



Algoritmos para Elección de Líder

Algoritmo de Bully

- Funciona en sistemas con un identificador único para cada nodo.
 - El nodo con el ID más alto se convierte en líder.
 - Si el líder falla, se inicia una nueva elección.
-
- **Ventaja:** Rápido y eficiente en redes pequeñas.
 - **Desventaja:** Genera mucho tráfico en redes grandes.



Funcionamiento del Algoritmo de Bully

1. Detección de fallo del líder: Un nodo detecta que el líder actual no responde (por ejemplo, debido a fallos en la red o hardware).
• Este nodo inicia el proceso de elección.

2. Inicio del proceso de elección: El nodo que detecta el fallo envía un mensaje ELECTION a todos los nodos con un identificador mayor al suyo.

3. Respuesta de los nodos superiores: Los nodos con identificadores mayores responden al mensaje ELECTION indicando que están disponibles.

- Si un nodo superior responde, el nodo que inició la elección abandona el proceso, ya que uno más “poderoso” se encargará de la elección.

4. Elección recursiva: El nodo superior que respondió comienza su propia ronda de elección, enviando mensajes ELECTION a nodos aún más altos.

5. Declaración del nuevo líder: Finalmente, el nodo con el **identificador más alto** no recibe respuesta de ningún nodo superior. Se proclama como líder y envía un mensaje COORDINATOR a todos los nodos, indicando que él es el nuevo líder.

Algoritmos para Elección de Líder

Algoritmo de Anillo (Ring Algorithm)

- Los nodos están organizados en un anillo lógico.
 - Si un nodo detecta que el líder ha fallado, inicia una elección circular.
-
- **Ventaja:** Reduce la sobrecarga de mensajes.
 - **Desventaja:** Puede ser lento en redes grandes.



Funcionamiento del Algoritmo de Anillo

1. Organización inicial: Los nodos se organizan en un anillo lógico, donde cada nodo tiene un sucesor conocido.

2. Inicio de la elección: Cuando un nodo detecta que el líder ha fallado, inicia una elección enviando un mensaje ELECTION con su propio identificador al siguiente nodo en el anillo.

3. Propagación del mensaje: Cada nodo que recibe el mensaje:

- Añade su propio identificador al mensaje.
- Reenvía el mensaje al siguiente nodo.

El mensaje continúa circulando por el anillo hasta que regresa al nodo que lo inició.

4. Determinación del líder: El nodo que inició la elección analiza los identificadores en el mensaje.

- Elige al nodo con el identificador más alto como líder.

5. Anuncio del líder: El nodo que inició la elección envía un mensaje COORDINATOR con el identificador del nuevo líder a través del anillo.

- Todos los nodos actualizan su estado para reconocer al nuevo líder.

Problema de Exclusión Mutua Distribuida

Cuando múltiples procesos en un sistema distribuido intentan acceder simultáneamente a un recurso compartido, es necesario un mecanismo para evitar colisiones. Este problema es conocido como **exclusión mutua distribuida**.

Ejemplo práctico

Imagina un sistema de reservas de boletos de avión:

- Dos usuarios intentan reservar el último asiento disponible.
- Si ambos procesos acceden al asiento al mismo tiempo, el sistema podría asignarlo dos veces, generando inconsistencias.

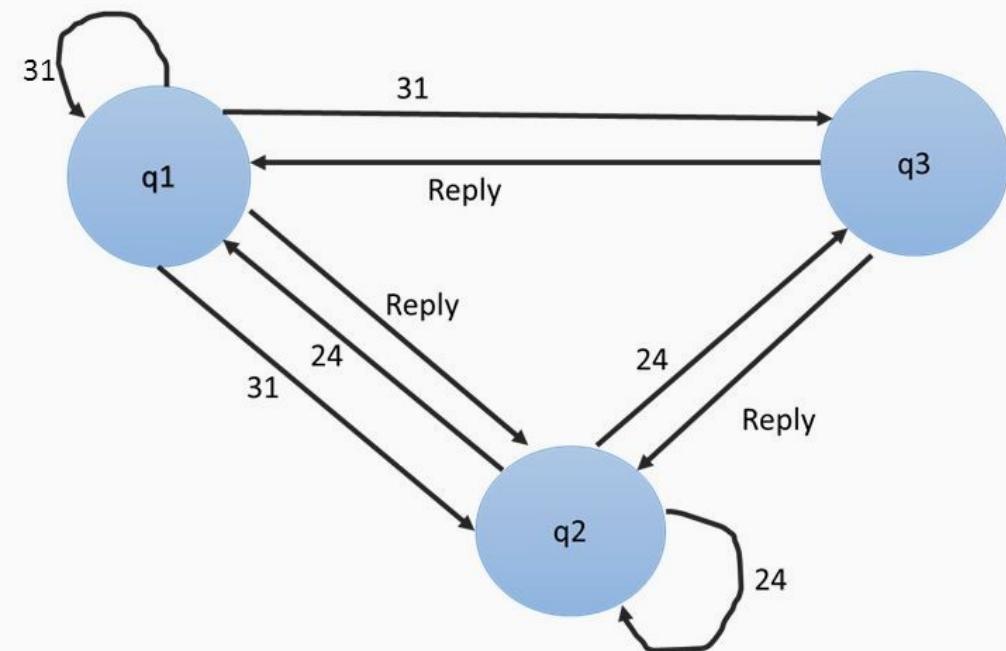


Algoritmo de Ricart-Agrawala

El algoritmo de Ricart-Agrawala es un enfoque distribuido para la exclusión mutua que elimina la necesidad de un nodo central. Está basado en el envío de mensajes entre los nodos participantes para garantizar que un solo proceso acceda al recurso compartido a la vez.

Funcionamiento

1. Cada nodo en el sistema tiene un identificador único.
2. Existe un reloj lógico para ordenar los eventos.
3. Los mensajes entre nodos son confiables y se entregan en el orden en que se enviaron.



Pasos del algoritmo:

1. Solicitud de acceso: Un nodo que desea acceder al recurso compartido genera una solicitud que incluye:

- Su identificador único.
- Una marca de tiempo generada por su reloj lógico.
- El nodo envía esta solicitud a todos los demás nodos.

2. Recepción de la solicitud: Al recibir una solicitud de acceso, un nodo tiene tres opciones:

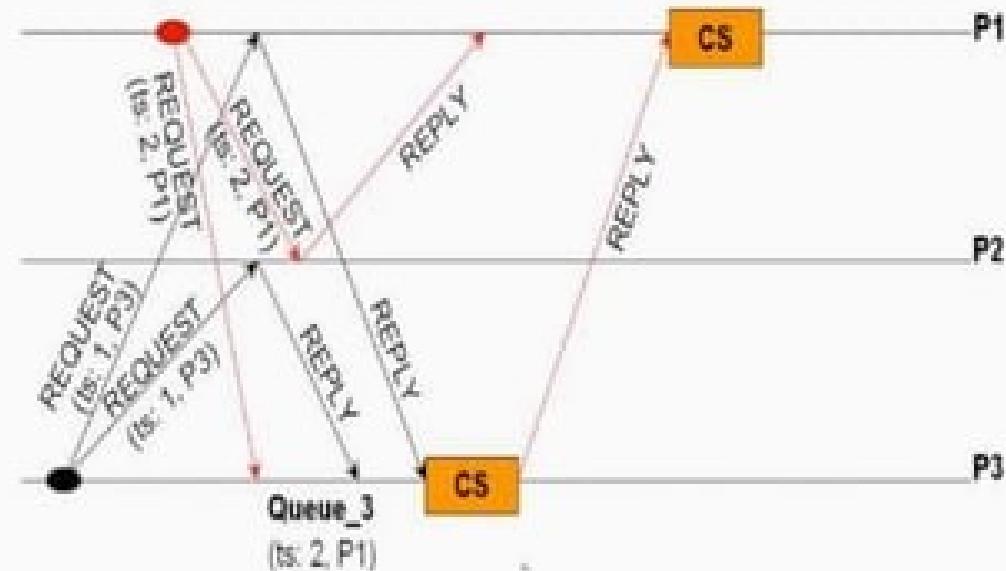
- **Si no está usando el recurso ni tiene solicitudes pendientes:** Responde inmediatamente con un mensaje de “OK”.
- **Si está usando el recurso compartido:** Retrasa su respuesta hasta liberar el recurso.
- **Si también desea acceder al recurso:** Compara las marcas de tiempo de las solicitudes:
 - La solicitud con la marca de tiempo más antigua tiene prioridad.
 - En caso de empate, el identificador del nodo decide.

Pasos del algoritmo:

3. Acceso al recurso: El nodo solicitante solo accede al recurso una vez que ha recibido respuestas “OK” de todos los demás nodos.

4. Liberación del recurso: Cuando el nodo termina de usar el recurso, envía un mensaje de “OK” a todos los nodos cuyas solicitudes estaban en espera.

Ricart-Agrawala: Example(2)

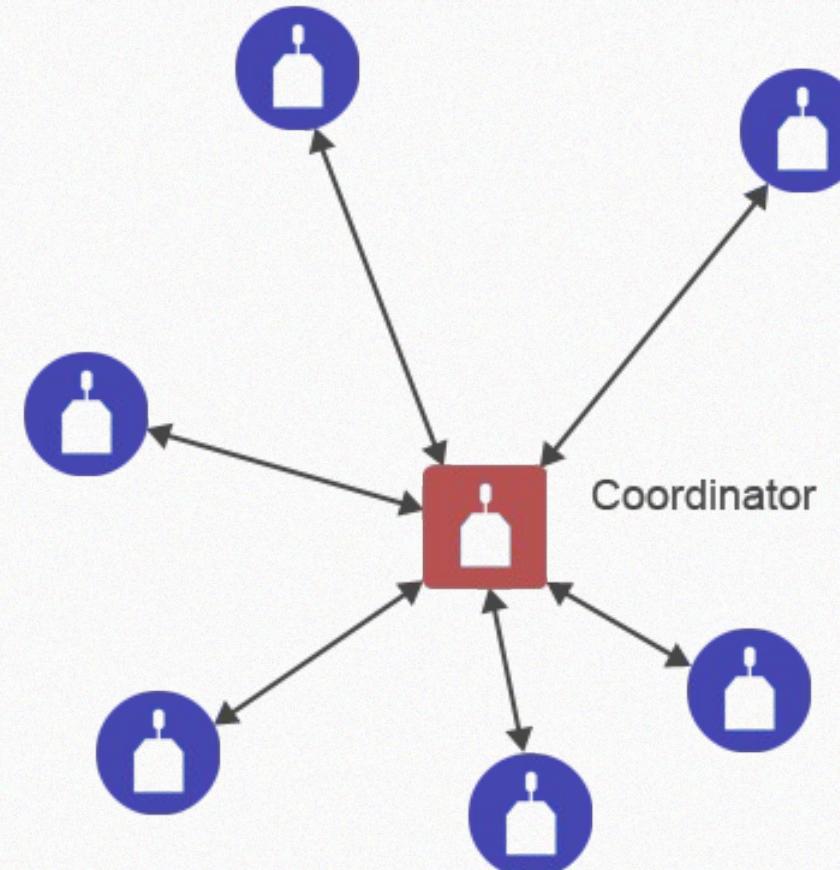


Algoritmo basado en un Coordinador Central

Este algoritmo utiliza un nodo central que actúa como árbitro. Todos los nodos que deseen acceder al recurso deben comunicarse con el coordinador para solicitar permiso.

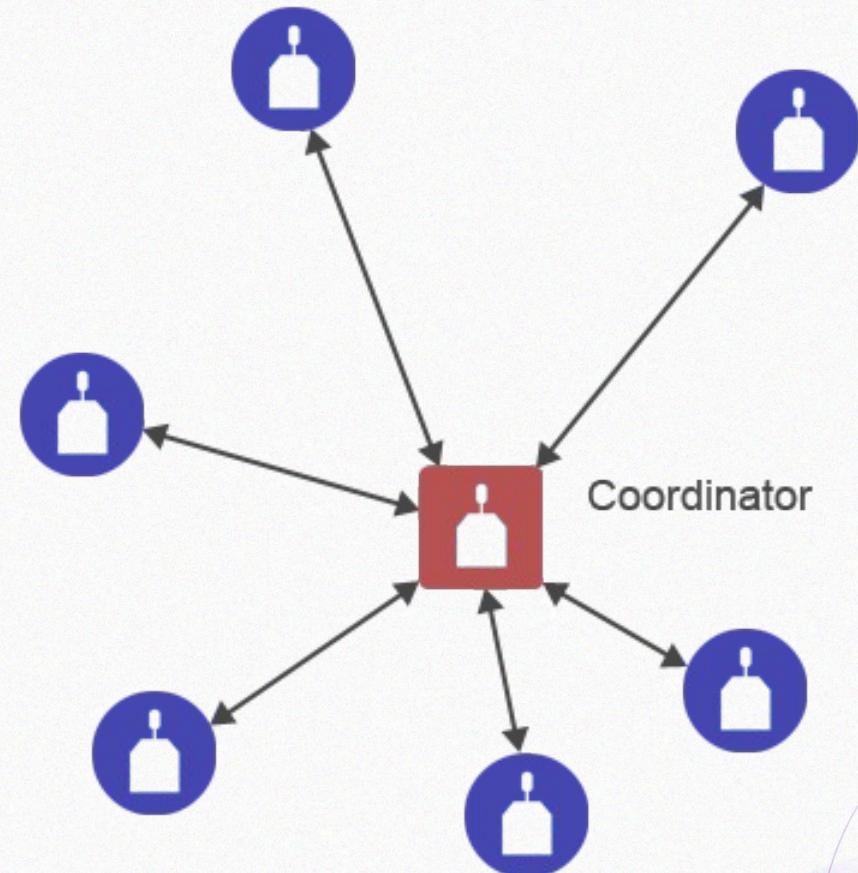
Supuestos:

1. Existe un nodo designado como **coordinador central**.
2. Todos los nodos participantes conocen la dirección del coordinador.



Pasos del algoritmo:

1. **Solicitud de acceso:** Un nodo que desea acceder al recurso envía un mensaje de solicitud al coordinador.
2. **Decisión del coordinador:**
 - Otorga el permiso si el recurso está disponible.
 - Coloca la solicitud en una cola si el recurso ya está en uso.
3. **Acceso al recurso:** El nodo solicitante usa el recurso una vez que recibe el permiso del coordinador.
4. **Liberación del recurso:** Cuando el nodo termina de usar el recurso, envía un mensaje de “liberación” al coordinador.
 - El coordinador otorga el acceso al siguiente nodo en la cola.



Tolerancia a Fallos

Tolerancia a Fallos en Sistemas Distribuidos

La **tolerancia a fallos** es la capacidad de un sistema para continuar operando correctamente incluso cuando ocurren fallos en sus componentes. Es un pilar clave en el diseño de sistemas distribuidos, ya que los fallos son inevitables en entornos complejos que involucran múltiples nodos, redes y dispositivos.



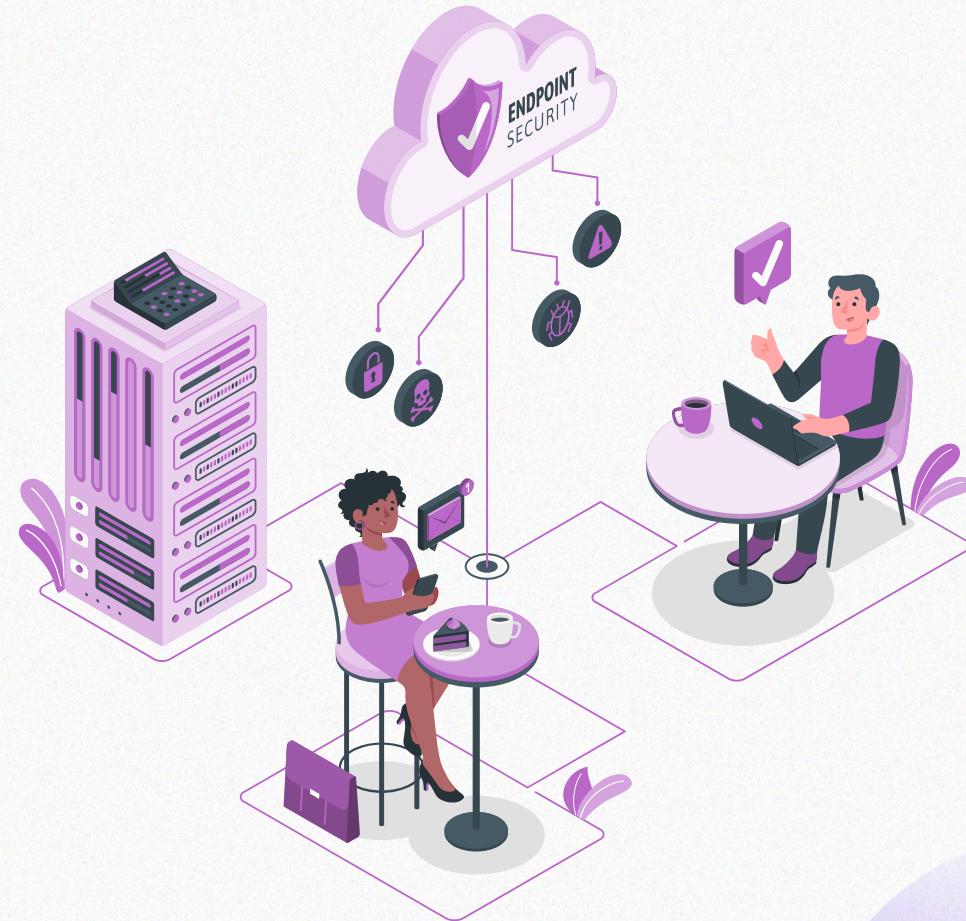
Tipos de Fallos

Fallos de Red

Estos fallos ocurren cuando la comunicación entre nodos en un sistema distribuido se interrumpe o degrada.

Ejemplos comunes:

- **Pérdida de paquetes:** Un mensaje no llega a su destino.
- **Latencia alta:** Los mensajes tardan demasiado en transmitirse.
- **Particiones de red:** Un grupo de nodos queda aislado del resto del sistema.



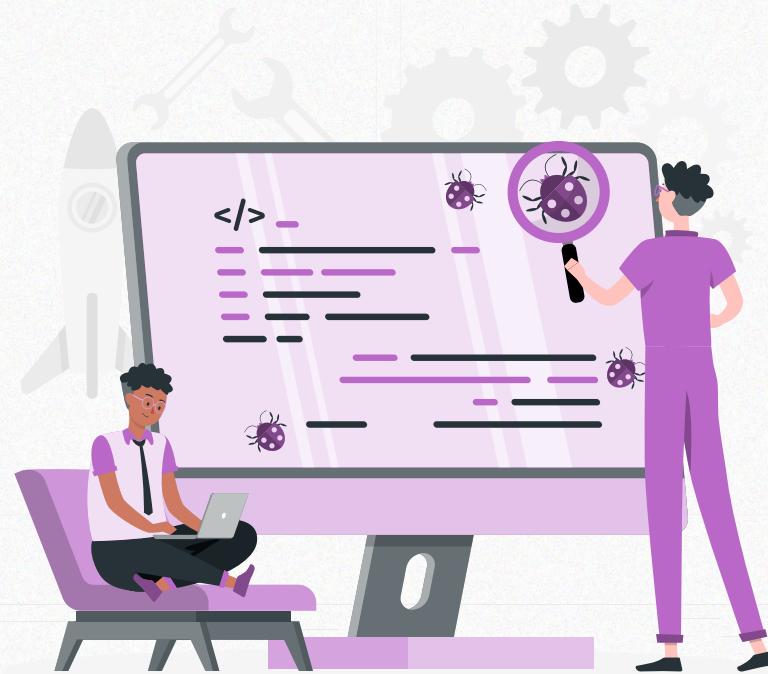
Tipos de Fallos

Fallos de Software

Los errores en el código o en la configuración del sistema también son una causa común de fallos.

Ejemplos comunes:

- Errores lógicos en el código (bugs).
- Configuración incorrecta del sistema.
- Actualizaciones fallidas.



Técnicas de Tolerancia a Fallos

Redundancia

La redundancia consiste en mantener múltiples copias de componentes o datos para asegurar la continuidad del servicio en caso de fallo.

Tipos de redundancia:

1. Redundancia de datos: Los datos se almacenan en múltiples nodos o discos.

- Ejemplo: RAID (Redundant Array of Independent Disks).

2. Redundancia de componentes: Sistemas que duplican hardware crítico como servidores o routers.

- Ejemplo: Un clúster de servidores donde si uno falla, otro toma su lugar.



Checkpointing

El **checkpointing** implica tomar instantáneas (snapshots) periódicas del estado de un sistema o proceso. Estas instantáneas permiten recuperar el estado previo en caso de un fallo.

Funcionamiento:

1. Durante la ejecución, el sistema guarda estados intermedios en disco o memoria.
2. Si ocurre un fallo, el sistema se reinicia desde el último estado guardado.



Replicación de Datos

La **replicación de datos** consiste en mantener múltiples copias de los datos en diferentes nodos. En caso de que un nodo falle, los datos pueden recuperarse desde otra copia.

1. **Síncrona:** Los datos se replican instantáneamente en todos los nodos.
 - **Ventaja:** Garantiza consistencia fuerte.
 - **Desventaja:** Introduce latencia.

2. **Asíncrona:** Los datos se replican con un retraso, después de confirmar la operación.
 - **Ventaja:** Mejora el rendimiento.
 - **Desventaja:** Puede provocar inconsistencias temporales.

Detectores de Fallos

Los detectores de fallos son mecanismos que supervisan continuamente los nodos en un sistema distribuido para identificar cuándo uno de ellos deja de responder.

Funcionamiento:

1. Cada nodo envía mensajes periódicos (heartbeats) para indicar que está activo.
2. Si un nodo no envía un mensaje en un intervalo definido, se asume que ha fallado.

Tipos de detectores:

1. **Detector confiable:** Nunca genera falsos positivos, pero puede tardar en detectar fallos.
2. **Detector rápido:** Detecta fallos rápidamente, pero puede generar falsos positivos.

Ejemplo de Tolerancia a Fallos: Sistema de Videoconferencia

Consideremos una plataforma de videoconferencia (como Zoom o Microsoft Teams) con servidores distribuidos globalmente.

- **Fallos de red:** Si un servidor pierde conectividad, otro servidor cercano asume la carga y los usuarios son redirigidos automáticamente.
- **Fallos de hardware:** Los datos de la sesión en curso (audio y video) se replican en tiempo real en múltiples servidores para evitar pérdida de información.
- **Checkpointing:** El estado de cada reunión (participantes, grabaciones, configuraciones) se guarda periódicamente para permitir la recuperación en caso de interrupción.
- **Detectores de fallos:** Supervisan constantemente los nodos y redirigen las conexiones si un servidor falla.

Algoritmos de Distribución de Carga

Introducción a la Distribución de Carga

En un sistema distribuido, las solicitudes de los clientes deben ser procesadas por múltiples servidores o nodos de cómputo.

Para evitar la sobrecarga de algunos servidores mientras otros están subutilizados, se aplican **algoritmos de balanceo de carga**.



Tipos de Balanceo de Carga

2.1 Algoritmos Estáticos: La asignación de carga se realiza antes de que el sistema comience a procesar solicitudes.

- No se adaptan a cambios en la carga en tiempo real.

Ejemplo: Round Robin.

2.2 Algoritmos Dinámicos: Reaccionan en tiempo real a cambios en la carga.

- Se requiere monitoreo constante del estado de los servidores.

Ejemplo: *Least Connections*.



Algoritmos Clásicos de Distribución de Carga

Round Robin

Cada solicitud se asigna secuencialmente al siguiente servidor en una lista circular.

- **Ventajas:** Simple, fácil de implementar.
- **Desventajas:** No considera la carga actual de los servidores.

```
class RoundRobinBalancer:  
    def __init__(self, servers):  
        self.servers = servers  
        self.index = 0  
  
    def get_server(self):  
        server = self.servers[self.index]  
        self.index = (self.index + 1) % len(self.servers)  
        return server  
  
# Ejemplo de uso  
servers = ["S1", "S2", "S3"]  
balancer = RoundRobinBalancer(servers)  
  
for _ in range(6):  
    print(balancer.get_server())
```

Algoritmos Clásicos de Distribución de Carga

Least Connections

Asigna nuevas solicitudes al servidor con menos conexiones activas.

Ventajas: Se adapta a la carga actual.

Desventajas: Necesita monitorear el estado de los servidores.

```
class LeastConnectionsBalancer:  
    def __init__(self, servers):  
        self.servers = {server: 0 for server in servers}  
  
    def get_server(self):  
        server = min(self.servers, key=self.servers.get)  
        self.servers[server] += 1  
        return server  
  
    def release_server(self, server):  
        self.servers[server] -= 1  
  
# Ejemplo de uso  
servers = ["S1", "S2", "S3"]  
balancer = LeastConnectionsBalancer(servers)  
  
for _ in range(6):  
    server = balancer.get_server()  
    print(f"Asignado a: {server}")  
    balancer.release_server(server)
```

Algoritmo de Consistent Hashing

Problema con el Hashing Tradicional

Si asignamos un servidor basado en `hash(request) % num_servers`, cualquier cambio en el número de servidores cambia la asignación de todos los datos.

```
class ConsistentHashBalancer:
    def __init__(self, servers, replicas=3):
        self.ring = {}
        self.sorted_keys = []
        self.replicas = replicas

    for server in servers:
        for i in range(replicas):
            key = self.hash_function(f"{server}-{i}")
            self.ring[key] = server
            self.sorted_keys.append(key)

    self.sorted_keys.sort()

def hash_function(self, key):
    return int(hashlib.md5(key.encode()).hexdigest(), 16) % (2**32)

def get_server(self, request):
    key = self.hash_function(request)
    index = bisect.bisect(self.sorted_keys, key) % len(self.sorted_keys)
    return self.ring[self.sorted_keys[index]]
```

Ejemplo de uso

```
servers = ["S1", "S2", "S3"]
balancer = ConsistentHashBalancer(servers)

for request in ["A", "B", "C", "D"]:
    print(f"Request {request} asignado a: {balancer.get_server(request)}")
```

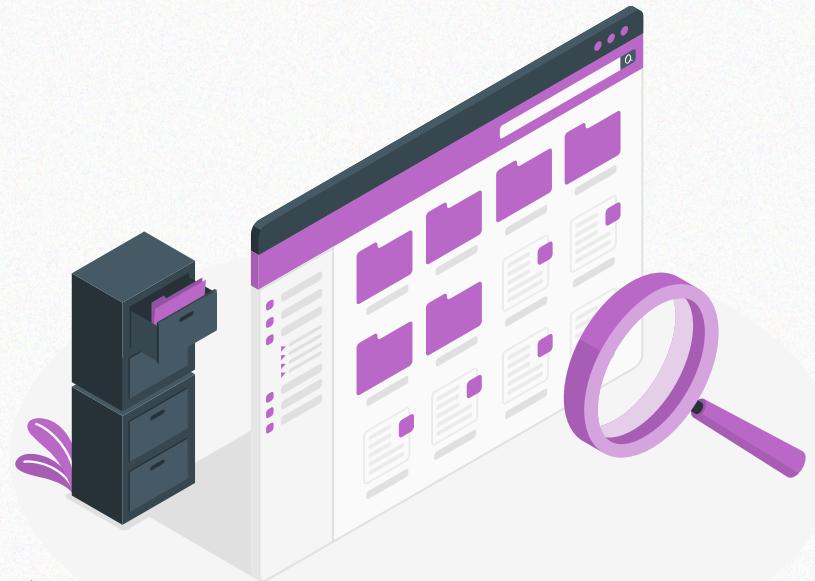
Algoritmos de Sistemas de Archivos Distribuidos

Introducción a los Sistemas de Archivos Distribuidos (DFS)

Un **sistema de archivos distribuido (DFS)** permite almacenar y acceder a archivos a través de múltiples nodos en una red, como si fueran parte de un solo sistema de archivos.

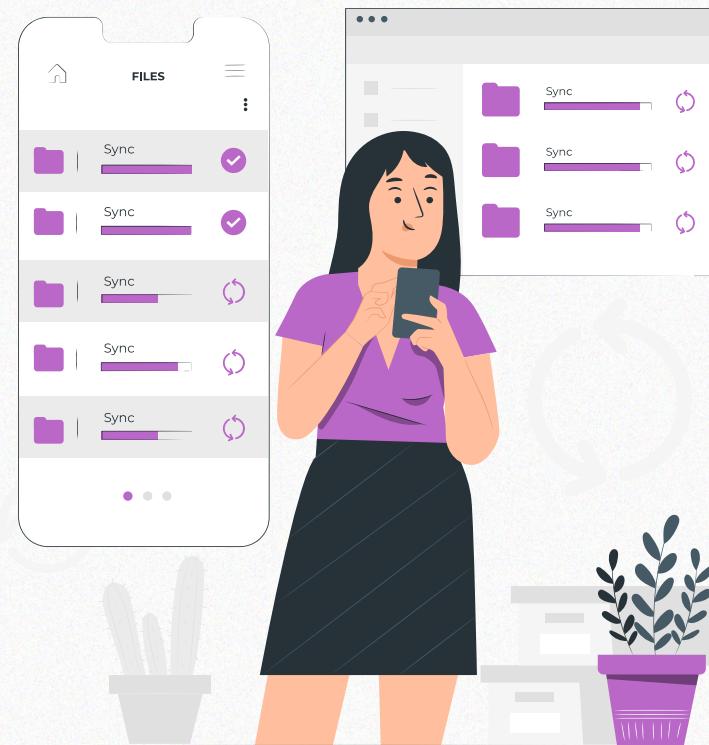
Ejemplos de DFS en la industria:

- **Google File System (GFS)** - Base de Hadoop HDFS.
- **Hadoop Distributed File System (HDFS)** - Procesamiento de Big Data.
- **Amazon S3** - Almacenamiento distribuido en la nube.



Características de un DFS

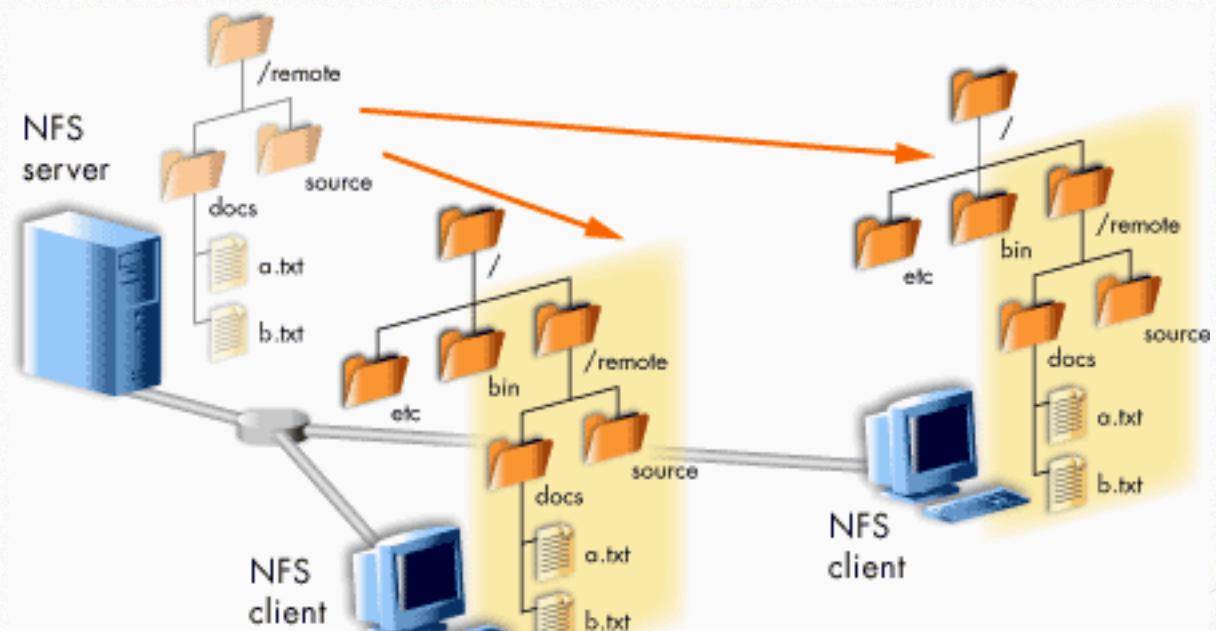
- **Transparencia de acceso:** Los usuarios no deben notar que los archivos están distribuidos.
- **Tolerancia a fallos:** Si un nodo falla, los datos deben permanecer accesibles.
- **Escalabilidad:** Debe manejar grandes volúmenes de datos y nodos.
- **Replicación y consistencia:** Múltiples copias de datos deben estar sincronizadas.



Arquitecturas de un DFS

2.1. Arquitectura Cliente-Servidor

- Un **servidor central** almacena metadatos (ubicación de archivos).
- Los clientes solicitan archivos y los servidores los proporcionan.
- Ejemplo: **NFS (Network File System)**.



rsync

Es una herramienta de sincronización de archivos en Linux que permite copiar y actualizar archivos entre directorios locales o remotos de manera eficiente.

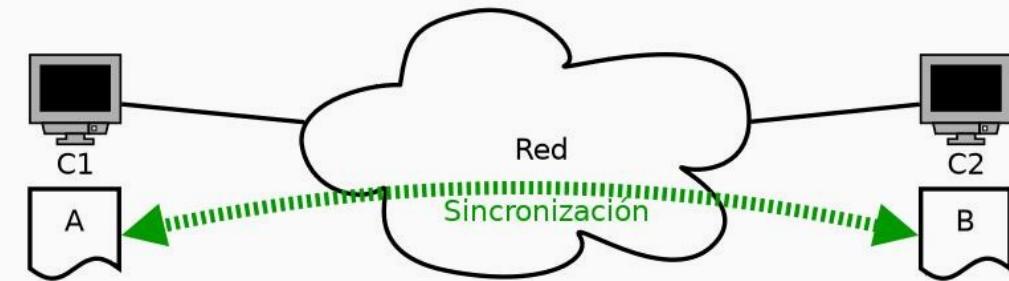
`rsync -av /origen /destino`

- `-a`: Modo de archivo (preserva permisos, enlaces, tím etc.).
- `-v`: Modo verboso (muestra detalles del proceso).

`rsync -avz /local/path usuario@servidor:/ruta/destino`

- `-z`: Comprime los datos durante la transferencia.

`rsync -avz usuario@servidor:/ruta/origen /local/path`

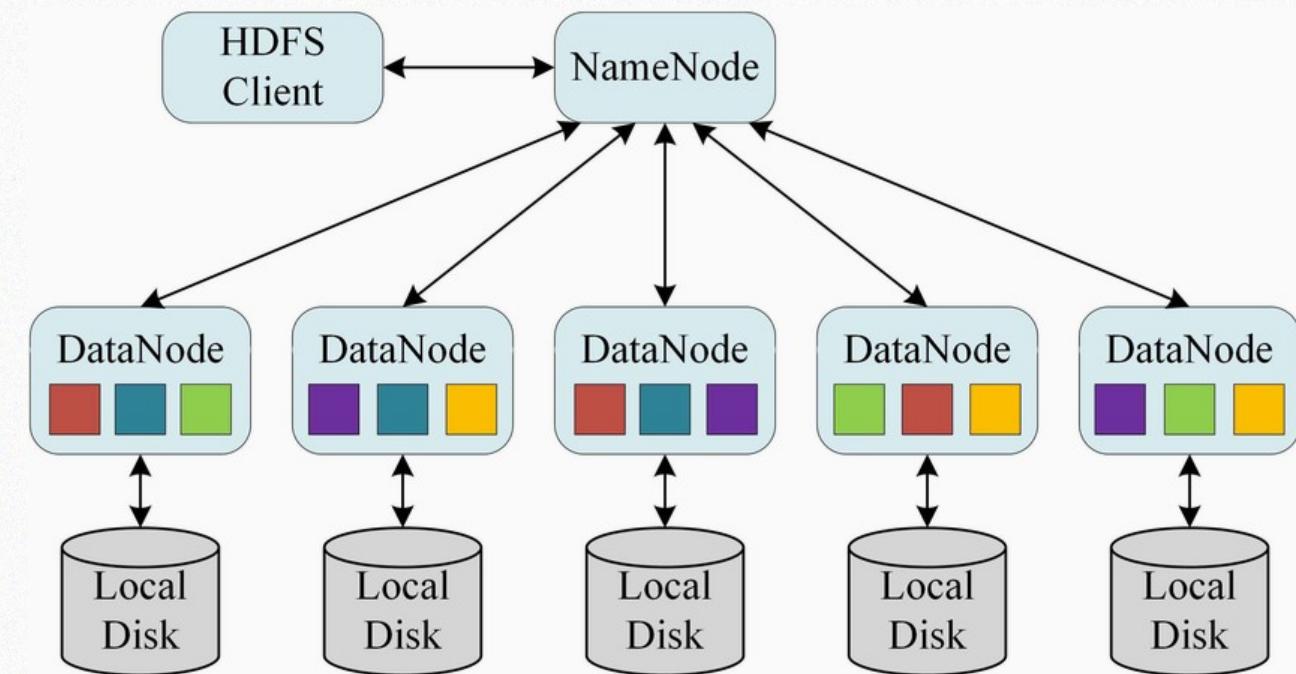


Arquitectura Basada en Bloques

HDFS: Hadoop Distributed File System

En HDFS, los archivos se dividen en fragmentos más pequeños llamados **bloques**. Cada uno de estos bloques es distribuido y replicado en varios nodos del clúster.

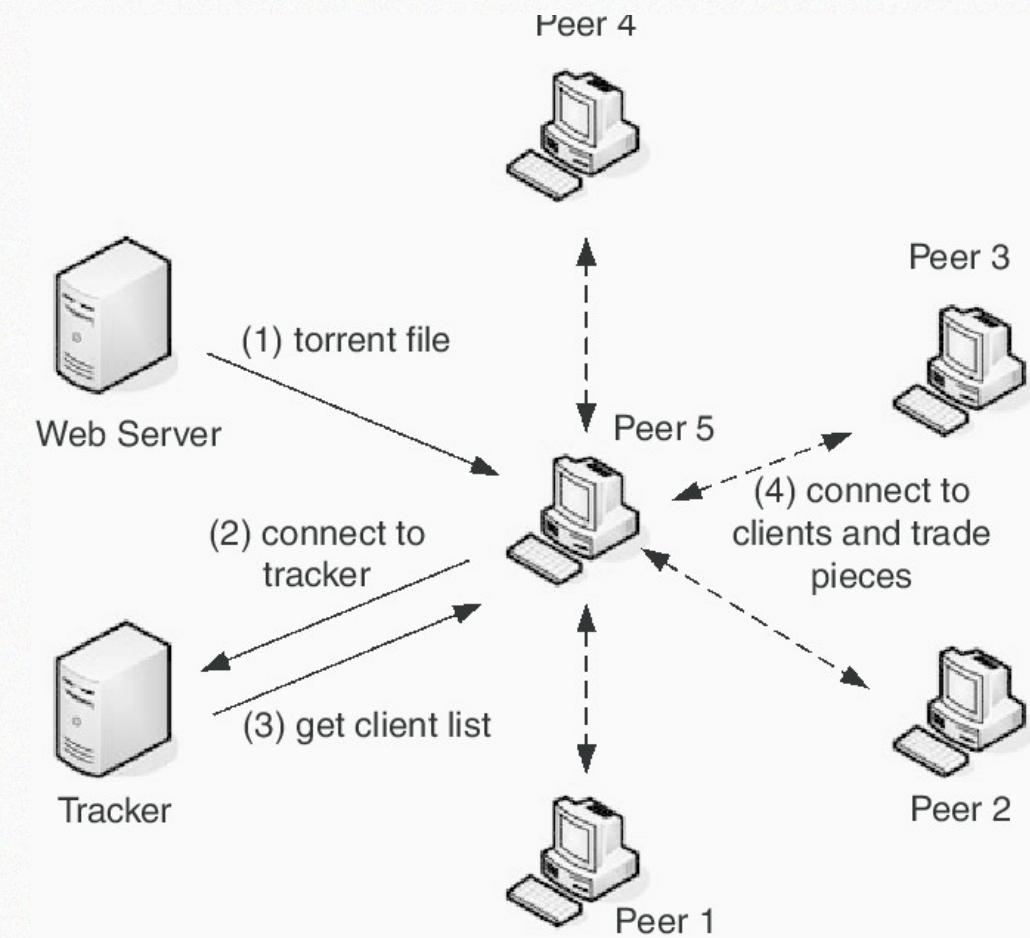
- División de archivos en bloques grandes (64MB, 128MB).
- Nodos DataNode almacenan bloques, un NameNode gestiona metadatos.
- Se usa en sistemas de **Big Data** y **procesamiento distribuido**.



Arquitectura Peer-to-Peer (P2P)

La **Arquitectura Peer-to-Peer (P2P)** es un modelo descentralizado en el que los nodos (peers) actúan simultáneamente como **clientes y servidores**, compartiendo recursos directamente entre sí sin la necesidad de un servidor centralizado.

En el contexto de **sistemas de archivos distribuidos**, esta arquitectura permite el almacenamiento y distribución de datos de manera eficiente, escalable y resistente a fallos.



Características de un Sistema de Archivos P2P

a) Descentralización

- No existe un único servidor que almacene toda la metadata o controle el sistema.
- Los archivos y sus fragmentos están distribuidos entre múltiples nodos de la red.

b) Escalabilidad

- Se adapta fácilmente al crecimiento del número de nodos sin afectar drásticamente el rendimiento.
- Más usuarios significan **más recursos disponibles**, lo que mejora la eficiencia del sistema.

c) Robustez y Tolerancia a Fallos

- La replicación distribuida hace que el sistema sea más resistente a caídas de nodos individuales.
- Si un nodo deja de estar disponible, otros pueden seguir proporcionando acceso al archivo.

d) Distribución de Carga

- El tráfico y la carga computacional se reparten entre los nodos en lugar de depender de un solo servidor.

Tipos de Arquitectura P2P en Sistemas de Archivos

a) P2P No Estructurado

- No sigue una organización fija.
- Los nodos se conectan de forma arbitraria y dependen de **broadcasts** o búsquedas aleatorias para encontrar archivos.
- Ejemplo: **Gnutella, Freenet**.

b) P2P Estructurado

- Usa algoritmos como **Distributed Hash Tables (DHTs)** para organizar y buscar archivos de manera eficiente.
- Cada nodo almacena partes específicas del índice global, lo que reduce la sobrecarga de búsqueda.
- Ejemplo: **BitTorrent (con trackers y DHT), Kademlia, Chord**.

c) P2P Híbrido

- Combina elementos de P2P y cliente-servidor.
- Utiliza servidores centrales solo para ciertas funciones (ej., facilitar la búsqueda), pero la transferencia de archivos ocurre directamente entre los peers.
- Ejemplo: **Napster, eMule (con servidores de búsqueda)**.

Algoritmos de Gestión de Datos en DFS

Fragmentación y Replicación

- **Fragmentación:** Divide un archivo en bloques pequeños.
- **Replicación:** Guarda copias en múltiples servidores.

```
class DistributedFileSystem:  
    def __init__(self, nodes):  
        self.nodes = {node: {} for node in nodes} # Simula almacenamiento  
  
    def store_file(self, filename, content, replicas=2):  
        """Almacena un archivo replicándolo en múltiples nodos"""  
        for i, node in enumerate(self.nodes.keys()):  
            if i >= replicas:  
                break  
            self.nodes[node][filename] = content  
        print(f"Archivo '{filename}' almacenado en {node}")  
  
    def retrieve_file(self, filename):  
        """Recupera un archivo desde cualquier nodo disponible"""  
        for node, files in self.nodes.items():  
            if filename in files:  
                print(f"Archivo '{filename}' recuperado desde {node}")  
                return files[filename]  
        return None  
  
# Simulación  
dfs = DistributedFileSystem(["Nodo1", "Nodo2", "Nodo3"])  
dfs.store_file("datos.txt", "Contenido importante")  
dfs.retrieve_file("datos.txt")
```

Algoritmos en Blockchain y Sistemas Descentralizados

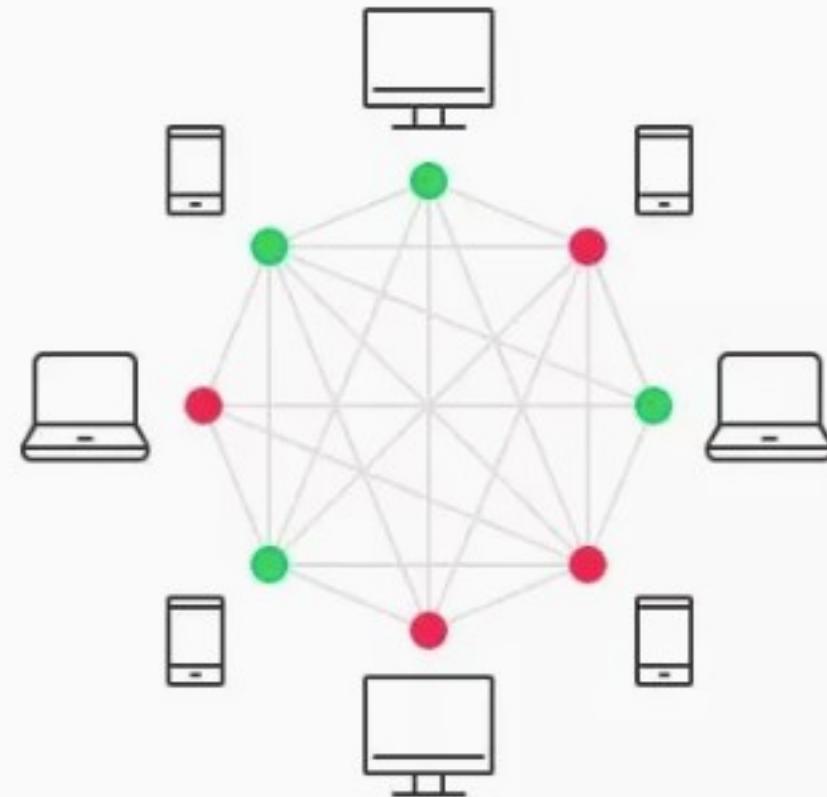
Introducción a los Sistemas Descentralizados

Los sistemas descentralizados no dependen de una única autoridad central.

En su lugar, las decisiones y el almacenamiento de datos se distribuyen entre múltiples nodos.

Ejemplos de sistemas descentralizados:

- **Blockchain** (Bitcoin, Ethereum).
- **Redes Peer-to-Peer (P2P)** (BitTorrent, IPFS).
- **Sistemas de archivos distribuidos** (IPFS, Filecoin).



¿Qué es Blockchain?

Blockchain es una **base de datos distribuida y descentralizada** que almacena información en bloques encadenados de manera segura mediante criptografía. Se usa principalmente para registrar transacciones de manera **segura, inmutable y transparente**.

Ejemplo Sencillo

Imagina un **libro contable digital** donde cada página es un bloque que almacena transacciones. Cada vez que una página se llena, se enlaza a la anterior y a la siguiente mediante un identificador único llamado **hash**. Este proceso garantiza que nadie puede modificar registros sin afectar toda la cadena.

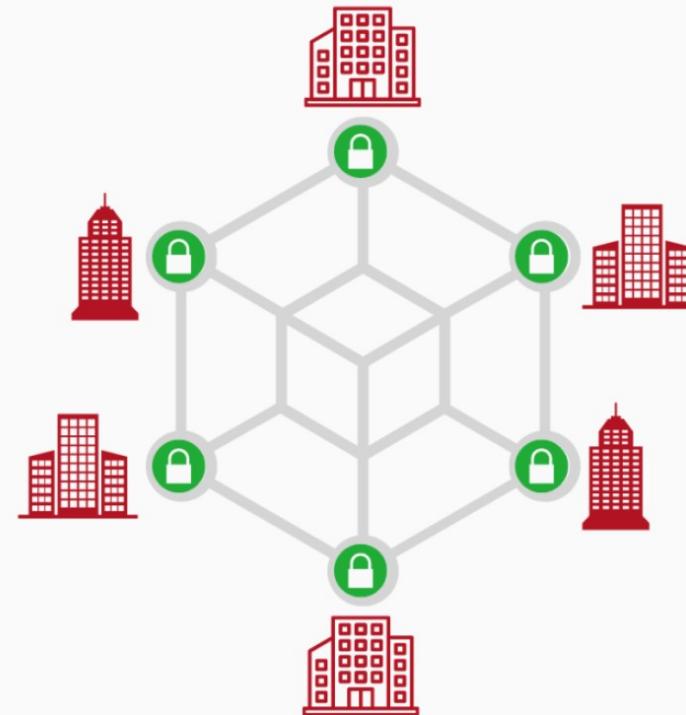


¿Cómo Funciona Blockchain?

Blockchain es una estructura de datos que almacena información en bloques encadenados de forma segura y descentralizada.

Características clave de Blockchain:

- **Inmutabilidad:** No se puede modificar información pasada.
- **Seguridad criptográfica:** Uso de *hashing* y firmas digitales.
- **Consenso distribuido:** Validación sin una autoridad central.



Estructura de un Bloque

Cada bloque contiene:

- **Datos de la transacción** (Ejemplo: envío de criptomonedas, contratos inteligentes, etc.).
- **Hash del bloque actual** (identificador único generado con criptografía).
- **Hash del bloque anterior**, formando una cadena.

```
{  
    "block_number": 12543,  
    "timestamp": "2025-01-29T14:35:00Z",  
    "transactions": [  
        {  
            "sender": "Alice",  
            "receiver": "Bob",  
            "amount": 2.5,  
            "currency": "BTC",  
            "transaction_id": "ab1234cd5678ef90"  
        },  
        {  
            "sender": "Carlos",  
            "receiver": "Diana",  
            "amount": 1.2,  
            "currency": "BTC",  
            "transaction_id": "cd9876ef5432ab10"  
        }  
    "previous_hash": "0000alb2c3d4e5f67890123456789abcdefabcdefabcde",  
    "current_hash": "0000f1e2d3c4b5a67890abcdefabcdefabcdefabcde",  
    "nonce": 294823,  
    "difficulty": 5  
}
```

Mecanismo de Consenso

Dado que Blockchain es **descentralizado**, necesita un mecanismo para validar transacciones sin un banco o entidad central. Existen varios mecanismos:

Proof of Work (PoW):

- Los mineros compiten para resolver un problema matemático complejo.
- Utilizado en Bitcoin.
- Alto consumo energético.

Proof of Stake (PoS):

- En lugar de mineros, hay validadores que apuestan sus criptomonedas para verificar transacciones.
- Más eficiente energéticamente.



Mecanismo de Consenso

Dado que Blockchain es **descentralizado**, necesita un mecanismo para validar transacciones sin un banco o entidad central. Existen varios mecanismos:

Proof of Work (PoW):

- Los mineros compiten para resolver un problema matemático complejo.
- Utilizado en Bitcoin.
- Alto consumo energético.

Proof of Stake (PoS):

- En lugar de mineros, hay validadores que apuestan sus criptomonedas para verificar transacciones.
- Más eficiente energéticamente.

Datos del Bloque

?

Hash con regla específica



Proof of Work (PoW)

El algoritmo usado por Bitcoin. Requiere que los mineros resuelvan un problema computacional difícil.

Pasos en PoW:

1. Un nodo selecciona transacciones y crea un bloque.
2. Se debe encontrar un *nonce* que produzca un hash con una cantidad de ceros iniciales.
3. El bloque es validado y agregado a la cadena.

```
def proof_of_work(block, difficulty=4):
    block.nonce = 0
    while block.compute_hash()[:difficulty] != "0" * difficulty:
        block.nonce += 1
    return block.compute_hash()

# Uso
block = Block(1, "00000", "Transacción 1")
mined_hash = proof_of_work(block)
print(f"Bloque minado con hash: {mined_hash}")
```

Proof of Stake (PoS)

Un método más eficiente que el PoW, utilizado en Ethereum 2.0.

Proof of Stake (PoS)

1. Un validador es seleccionado en función de su participación (*stake*).
2. El validador certifica transacciones y recibe recompensas.

```
import random

class ProofOfStake:
    def __init__(self):
        self.stakers = {}

    def add_staker(self, address, amount):
        self.stakers[address] = amount

    def select_validator(self):
        return random.choices(list(self.stakers.keys()), we

# Uso
pos = ProofOfStake()
pos.add_staker("Alice", 10)
pos.add_staker("Bob", 20)
print(f"Validador seleccionado: {pos.select_validator()}")
```

Nodos

Son las computadoras que participan en la red y almacenan una copia del Blockchain.

Tipos de nodos:

- **Nodo completo:** Guarda una copia completa de toda la Blockchain.
- **Nodo ligero:** Solo descarga información necesaria para transacciones específicas.



¿Cómo Funciona una Transacción en Blockchain?

Ejemplo: Enviar 1 Bitcoin de Alice a Bob

1. Alice inicia la transacción en su wallet digital.
2. La transacción se transmite a la red de nodos.
3. Mineros o validadores verifican la transacción con el mecanismo de consenso.
4. Si es válida, se agrupa en un bloque con otras transacciones.
5. El bloque se agrega a la cadena y la transacción es confirmada.
6. Bob recibe su 1 Bitcoin.



Algoritmos de Propagación de Información en Blockchain

Algoritmo de Gossip

Cómo funciona:

- Un nodo comparte nueva información con un subconjunto de vecinos.
- Los vecinos propagan la información a otros nodos hasta que toda la red la tenga.

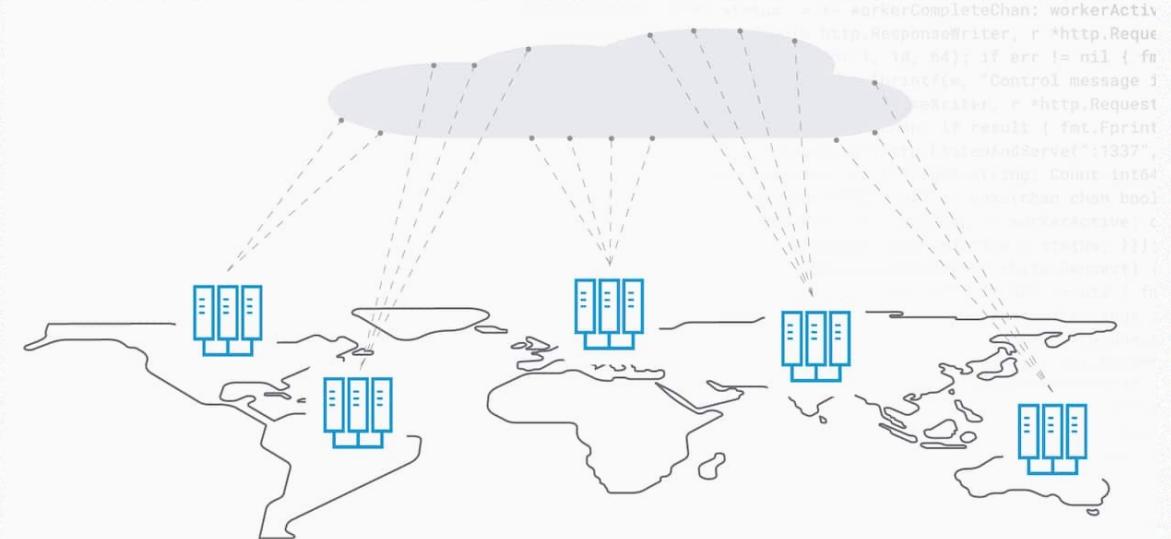
```
class GossipNetwork:  
    def __init__(self, nodes):  
        self.nodes = {node: [] for node in nodes}  
  
    def propagate(self, sender, message):  
        selected_nodes = random.sample(self.nodes.keys(), 2)  
        for node in selected_nodes:  
            self.nodes[node].append(message)  
            print(f"{sender} → {node}: {message}")  
  
# Uso  
network = GossipNetwork(["Nodo1", "Nodo2", "Nodo3", "Nodo4"])  
network.propagate("Nodo1", "Nuevo bloque")
```

Algoritmos de Computación Distribuida en la Nube

Introducción a la Computación Distribuida en la Nube

La computación distribuida es un paradigma en el que múltiples sistemas informáticos trabajan juntos de manera coordinada para resolver problemas computacionales. En este enfoque, las tareas se dividen entre múltiples nodos interconectados, cada uno de los cuales ejecuta una parte del procesamiento.

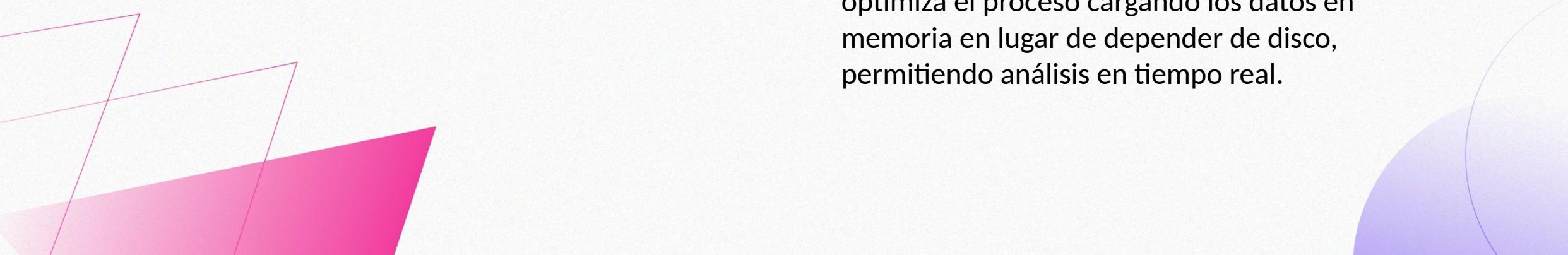
La computación en la nube aprovecha la computación distribuida para ofrecer servicios escalables, de alta disponibilidad y resilientes.



Ejemplos de Computación Distribuida en la Nube

Big Data Processing: Hadoop y Apache Spark

Imagina que una empresa como **Twitter** necesita analizar miles de millones de tweets diariamente para detectar tendencias y recomendar contenido.

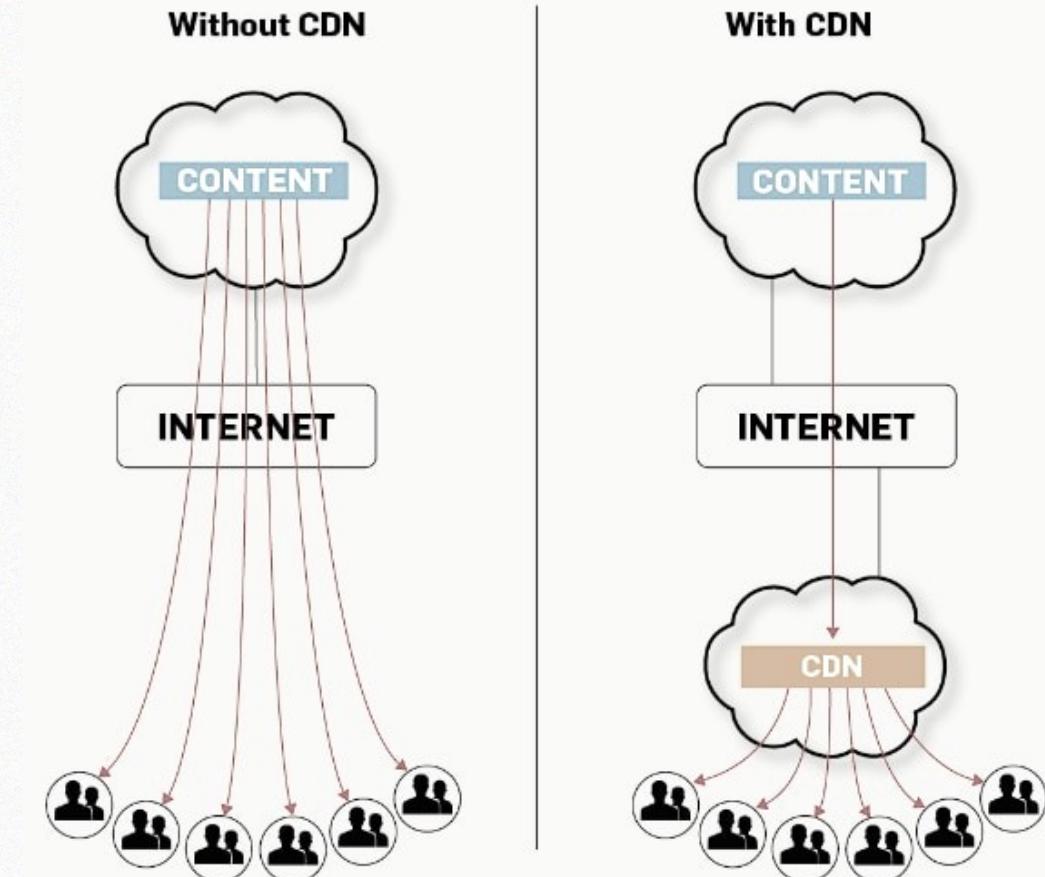


- **Hadoop HDFS (Hadoop Distributed File System):** Almacena grandes volúmenes de datos en múltiples nodos de almacenamiento en la nube.
- **MapReduce:** Divide la tarea en múltiples unidades más pequeñas, asignándolas a distintos nodos para su procesamiento en paralelo.
- **Apache Spark:** A diferencia de Hadoop, Spark optimiza el proceso cargando los datos en memoria en lugar de depender de disco, permitiendo análisis en tiempo real.

Ejemplos de Computación Distribuida en la Nube

Content Delivery Networks (CDN): Cloudflare y AWS CloudFront

Las CDN son redes distribuidas de servidores que almacenan copias en caché de contenido estático y dinámico en diferentes ubicaciones geográficas. Esto permite reducir la latencia y mejorar la velocidad de entrega de contenido.



Aplicaciones y Casos de Uso de la Computación Distribuida en la Nube

Procesamiento de Video en la Nube: Netflix y su Infraestructura Distribuida

Netflix es uno de los principales ejemplos de procesamiento de video en la nube. Maneja miles de petabytes de datos y distribuye contenido a millones de usuarios globalmente.



Cómo funciona el procesamiento distribuido en Netflix:

1. Codificación y Transcodificación de Video:

Cuando Netflix agrega una nueva película o serie, los archivos originales deben convertirse a múltiples formatos y resoluciones (480p, 720p, 1080p, 4K).

- Se usan **clusters de procesamiento distribuido** en AWS con tecnologías como **FFmpeg** y Amazon **Elastic Transcoder**.
- Cada segmento de video se procesa en **paralelo** en distintos servidores de la nube para acelerar el proceso.

2. Distribución de Video a Tráves de CDN: Netflix utiliza **AWS CloudFront** y **Open Connect** (su propia red de servidores perimetrales).

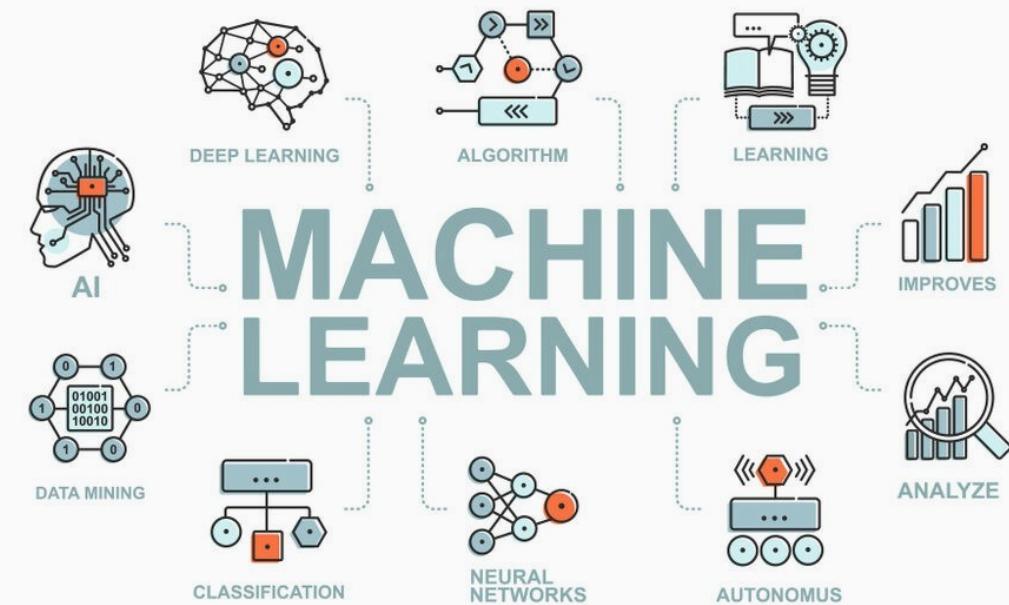
- Los servidores CDN almacenan copias de los videos en **múltiples regiones** para reducir la latencia.
- Cuando un usuario presiona “Reproducir”, la película no se descarga desde un único centro de datos, sino desde **el nodo más cercano** a su ubicación.

3. Optimización del Streaming con Computación Distribuida: La nube ajusta **dinámicamente la calidad del video** según el ancho de banda disponible (Adaptive Bitrate Streaming).

- Servidores distribuidos en la nube analizan en tiempo real el tráfico y la demanda, redirigiendo usuarios automáticamente para evitar saturaciones.

Machine Learning Distribuido: TensorFlow en la Nube

El entrenamiento de modelos de Machine Learning consume enormes cantidades de cómputo. Para reducir el tiempo de entrenamiento, se usa **computación distribuida en la nube** con frameworks como **TensorFlow** y **PyTorch**.



Reconocimiento Facial en la Nube (Google Photos)

1. Carga de Datos en la Nube:

- Google Photos sube imágenes de millones de usuarios a **Google Cloud Storage**.
- Las fotos se almacenan en formato distribuido en bases de datos como **BigQuery** o **Firestore**.

2. Entrenamiento Distribuido del Modelo:

- Google usa **TPUs (Tensor Processing Units)**, que son chips optimizados para IA.
- Los datos se dividen en lotes y se distribuyen a múltiples **nodos de entrenamiento** en la nube.
- Cada nodo entrena una parte del modelo, y luego se combinan los resultados en un proceso llamado **reducción colectiva** (AllReduce).

3. Predicción en Tiempo Real:

- Cuando subes una foto, el modelo entrenado se ejecuta en un servidor distribuido.
- **Google Cloud AI** reconoce automáticamente **rostros y objetos** en tus fotos en milisegundos.